1.1 ./csharp/Platform.Data.Doublets/CriterionMatchers/TargetMatcher.cs

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

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

namespace Platform.Data.Doublets.CriterionMatchers
{
    public class TargetMatcher<TLink> : LinksOperatorBase<TLink>, ICriterionMatcher<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly TLink _targetToMatch;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TargetMatcher(ILinks<TLink> links, TLink targetToMatch) : base(links) =>
        ↪  _targetToMatch = targetToMatch;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsMatched(TLink link) => _equalityComparer.Equals(_links.GetTarget(link),
        ↪  _targetToMatch);
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

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);
        }
    }
}
```

1.3 ./csharp/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);
        }
    }
}
```

1.4 ./csharp/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>
    {
        protected readonly LinksConstants<TLink> _constants;

        public LinksConstants<TLink> Constants
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _constants;
        }

        protected ILinks<TLink> _facade;

        public ILinks<TLink> Facade
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _facade;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set
            {
                _facade = value;
                if (_links is LinksDecoratorBase<TLink> decorator)
                {
                    decorator.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);
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1063 // Implement IDisposable Correctly

namespace Platform.Data.Doublets.Decorators
{
    public abstract class LinksDisposableDecoratorBase<TLink> : LinksDecoratorBase<TLink>,
    ↪  ILinks<TLink>, System.IDisposable
    {
        protected class DisposableWithMultipleCallsAllowed : Disposable
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public DisposableWithMultipleCallsAllowed(Disposal disposal) : base(disposal) { }

            protected override bool AllowMultipleDisposeCalls
```

```
            {
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                get => true;
            }
        }

        protected readonly DisposableWithMultipleCallsAllowed Disposable;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksDisposableDecoratorBase(ILinks<TLink> links) : base(links) => Disposable
        ↪  = new DisposableWithMultipleCallsAllowed(Dispose);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        ~LinksDisposableDecoratorBase() => Disposable.Destruct();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Dispose() => Disposable.Dispose();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                _links.DisposeIfPossible();
            }
        }
    }
}
```

1.6 ./csharp/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)
        {
            var links = _links;
            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)
            var links = _links;
            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];
            var links = _links;
            links.EnsureLinkExists(link, nameof(link));
            links.Delete(link);
        }
    }
}
```

1.7 ./csharp/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;
            if (!_equalityComparer.Equals(constants.Any, itselfConstant) &&
            ↪  restrictions.Contains(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));
    }
}
```

1.8 ./csharp/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;
            var links = _links;
            links.EnsureCreated(substitution[constants.SourcePart],
            ↪  substitution[constants.TargetPart]);
            return links.Update(restrictions, substitution);
        }
    }
}
```

1.9 ./csharp/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)]
```

```
14          public override TLink Create(IList<TLink> restrictions) => _links.CreatePoint();
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
                ↪ _links.Update(restrictions, _links.ResolveConstantAsSelfReference(_constants.Null,
                ↪ restrictions, substitution));
18      }
19  }
```

1.10 ./csharp/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 constants = _constants;
19              var links = _links;
20              var newLinkAddress = links.SearchOrDefault(substitution[constants.SourcePart],
                    ↪ substitution[constants.TargetPart]);
21              if (_equalityComparer.Equals(newLinkAddress, default))
22              {
23                  return links.Update(restrictions, substitution);
24              }
25              return ResolveAddressChangeConflict(restrictions[constants.IndexPart],
                    ↪ newLinkAddress);
26          }
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
                ↪ newLinkAddress)
30          {
31              if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
                    ↪ _links.Exists(oldLinkAddress))
32              {
33                  _facade.Delete(oldLinkAddress);
34              }
35              return newLinkAddress;
36          }
37      }
38  }
```

1.11 ./csharp/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              var links = _links;
17              var constants = _constants;
18              links.EnsureDoesNotExists(substitution[constants.SourcePart],
                    ↪ substitution[constants.TargetPart]);
19              return links.Update(restrictions, substitution);
20          }
21      }
22  }
```

1.12 ./csharp/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)
        {
            var links = _links;
            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];
            var links = _links;
            links.EnsureNoUsages(link);
            links.Delete(link);
        }
    }
}
```

1.13 ./csharp/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];
            var links = _links;
            links.EnforceResetValues(linkIndex);
            links.Delete(linkIndex);
        }
    }
}
```

1.14 ./csharp/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>
    /// <para>Represents a combined decorator that implements the basic logic for interacting
    ↪  with the links storage for links with addresses represented as <see cref="System.UInt64"
    ↪  />.</para>
    /// <para>Представляет комбинированный декоратор, реализующий основную логику по
    ↪  взаимодействии с хранилищем связей, для связей с адресами представленными в виде <see
    ↪  cref="System.UInt64"/>.</para>
    /// </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)
```

```
19      ///     + меньше объём БД
20      ///     - больше сложность
21      ///
22      /// Текущее теоретическое ограничение на индекс связи, из-за использования 5 бит в размере
        ↪  поддеревьев для AVL баланса и флагов нитей: 2 в степени(64 минус 5 равно 59 ) равно 576
        ↪  460 752 303 423 488
23      /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
        ↪  (битовыми строками) - вариант матрицы (выстраеваемой лениво).
24      ///
25      /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
        ↪  выбрасываться только при #if DEBUG
26      /// </remarks>
27      public class UInt64Links : LinksDisposableDecoratorBase<ulong>
28      {
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          public UInt64Links(ILinks<ulong> links) : base(links) { }
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          public override ulong Create(IList<ulong> restrictions) => _links.CreatePoint();
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
37          {
38              var constants = _constants;
39              var indexPartConstant = constants.IndexPart;
40              var sourcePartConstant = constants.SourcePart;
41              var targetPartConstant = constants.TargetPart;
42              var nullConstant = constants.Null;
43              var itselfConstant = constants.Itself;
44              var existedLink = nullConstant;
45              var updatedLink = restrictions[indexPartConstant];
46              var newSource = substitution[sourcePartConstant];
47              var newTarget = substitution[targetPartConstant];
48              var links = _links;
49              if (newSource != itselfConstant && newTarget != itselfConstant)
50              {
51                  existedLink = links.SearchOrDefault(newSource, newTarget);
52              }
53              if (existedLink == nullConstant)
54              {
55                  var before = links.GetLink(updatedLink);
56                  if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
                    ↪  newTarget)
57                  {
58                      links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
                        ↪  newSource,
59                                                newTarget == itselfConstant ? updatedLink :
                                                  ↪  newTarget);
60                  }
61                  return updatedLink;
62              }
63              else
64              {
65                  return _facade.MergeAndDelete(updatedLink, existedLink);
66              }
67          }
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          public override void Delete(IList<ulong> restrictions)
71          {
72              var linkIndex = restrictions[_constants.IndexPart];
73              var links = _links;
74              links.EnforceResetValues(linkIndex);
75              _facade.DeleteAllUsages(linkIndex);
76              links.Delete(linkIndex);
77          }
78      }
79  }
```

1.15 ./csharp/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
```

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

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

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

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

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

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

```
////                    var matchDecision = matchedHandler(matchedLink, newValue);
////                    if (Equals(matchDecision, Constants.Break))
////                        return false;
////                    if (!Equals(matchDecision, Constants.Skip))
////                        transitions.Add(new Transition(matchedLink, newValue));
////                    return true;
////                };
////                if (!Memory.Each(handler, restriction))
////                    return Constants.Break;
////            }
////        }
////        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)
//    {
```

```
150             //        matchedLinks = new List<IList<T>>();
151             //        Func<T, bool> handler = link =>
152             //        {
153             //            var matchedLink = Memory.GetLinkValue(link);
154             //            var matchDecision = matchedHandler(matchedLink);
155             //            if (Equals(matchDecision, Constants.Break))
156             //                return false;
157             //            if (!Equals(matchDecision, Constants.Skip))
158             //                matchedLinks.Add(matchedLink);
159             //            return true;
160             //        };
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             var constants = _constants;
196             if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
197             {
198                 return constants.Continue;
199             }
200             else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
                ↪  Check if it is a correct condition
201             {
202                 // Or it only applies to trigger without matchHandler.
203                 throw new NotImplementedException();
204             }
205             else if (!substitution.IsNullOrEmpty()) // Creation
206             {
207                 var before = Array.Empty<TLink>();
208                 // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
                    ↪  (пройти мимо) или пустить (взять)?
209                 if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    ↪  constants.Break))
210                 {
211                     return constants.Break;
212                 }
213                 var after = (IList<TLink>)substitution.ToArray();
214                 if (_equalityComparer.Equals(after[0], default))
215                 {
216                     var newLink = _links.Create();
217                     after[0] = newLink;
218                 }
219                 if (substitution.Count == 1)
```

```
220                     {
221                         after = _links.GetLink(substitution[0]);
222                     }
223                     else if (substitution.Count == 3)
224                     {
225                         //Links.Create(after);
226                     }
227                     else
228                     {
229                         throw new NotSupportedException();
230                     }
231                     if (matchHandler != null)
232                     {
233                         return substitutionHandler(before, after);
234                     }
235                     return constants.Continue;
236                 }
237                 else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
238                 {
239                     if (patternOrCondition.Count == 1)
240                     {
241                         var linkToDelete = patternOrCondition[0];
242                         var before = _links.GetLink(linkToDelete);
243                         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                            ↪ constants.Break))
244                         {
245                             return constants.Break;
246                         }
247                         var after = Array.Empty<TLink>();
248                         _links.Update(linkToDelete, constants.Null, constants.Null);
249                         _links.Delete(linkToDelete);
250                         if (matchHandler != null)
251                         {
252                             return substitutionHandler(before, after);
253                         }
254                         return constants.Continue;
255                     }
256                     else
257                     {
258                         throw new NotSupportedException();
259                     }
260                 }
261                 else // Replace / Update
262                 {
263                     if (patternOrCondition.Count == 1) //-V3125
264                     {
265                         var linkToUpdate = patternOrCondition[0];
266                         var before = _links.GetLink(linkToUpdate);
267                         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                            ↪ constants.Break))
268                         {
269                             return constants.Break;
270                         }
271                         var after = (IList<TLink>)substitution.ToArray(); //-V3125
272                         if (_equalityComparer.Equals(after[0], default))
273                         {
274                             after[0] = linkToUpdate;
275                         }
276                         if (substitution.Count == 1)
277                         {
278                             if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
279                             {
280                                 after = _links.GetLink(substitution[0]);
281                                 _links.Update(linkToUpdate, constants.Null, constants.Null);
282                                 _links.Delete(linkToUpdate);
283                             }
284                         }
285                         else if (substitution.Count == 3)
286                         {
287                             //Links.Update(after);
288                         }
289                         else
290                         {
291                             throw new NotSupportedException();
292                         }
293                         if (matchHandler != null)
294                         {
295                             return substitutionHandler(before, after);
```

```
                    }
                    return constants.Continue;
                }
                else
                {
                    throw new NotSupportedException();
                }
            }
        }

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

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

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

        public T Source
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }
        public T Target
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Doublet(T source, T target)
        {
            Source = source;
            Target = target;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override string ToString() => $"{Source}->{Target}";

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
        ↪  && _equalityComparer.Equals(Target, other.Target);
```

```

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator ==(Doublet<T> left, Doublet<T> right) => left.Equals(right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator !=(Doublet<T> left, Doublet<T> right) => !(left == right);
    }
}
```

1.17 ./csharp/Platform.Data.Doublets/DoubletComparer.cs

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

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

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

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

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

1.18 ./csharp/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>>
    {
    }
}
```

1.19 ./csharp/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.Random;
using Platform.Setters;
using Platform.Converters;
using Platform.Numbers;
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
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void RunRandomCreations<TLink>(this ILinks<TLink> links, ulong
        ↪  amountOfCreations)
        {
            var random = RandomHelpers.Default;
            var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
            var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
```

```
27              for (var i = 0UL; i < amountOfCreations; i++)
28              {
29                  var linksAddressRange = new Range<ulong>(0,
                    ↪  addressToUInt64Converter.Convert(links.Count()));
30                  var source =
                    ↪  uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
31                  var target =
                    ↪  uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
32                  links.GetOrCreate(source, target);
33              }
34          }
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          public static void RunRandomSearches<TLink>(this ILinks<TLink> links, ulong
            ↪  amountOfSearches)
38          {
39              var random = RandomHelpers.Default;
40              var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
41              var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
42              for (var i = 0UL; i < amountOfSearches; i++)
43              {
44                  var linksAddressRange = new Range<ulong>(0,
                    ↪  addressToUInt64Converter.Convert(links.Count()));
45                  var source =
                    ↪  uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
46                  var target =
                    ↪  uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
47                  links.SearchOrDefault(source, target);
48              }
49          }
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, ulong
            ↪  amountOfDeletions)
53          {
54              var random = RandomHelpers.Default;
55              var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
56              var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
57              var linksCount = addressToUInt64Converter.Convert(links.Count());
58              var min = amountOfDeletions > linksCount ? 0UL : linksCount - amountOfDeletions;
59              for (var i = 0UL; i < amountOfDeletions; i++)
60              {
61                  linksCount = addressToUInt64Converter.Convert(links.Count());
62                  if (linksCount <= min)
63                  {
64                      break;
65                  }
66                  var linksAddressRange = new Range<ulong>(min, linksCount);
67                  var link =
                    ↪  uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
68                  links.Delete(link);
69              }
70          }
71
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          public static void Delete<TLink>(this ILinks<TLink> links, TLink linkToDelete) =>
            ↪  links.Delete(new LinkAddress<TLink>(linkToDelete));
74
75          /// <remarks>
76          /// TODO: Возможно есть очень простой способ это сделать.
77          /// (Например просто удалить файл, или изменить его размер таким образом,
78          /// чтобы удалился весь контент)
79          /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
80          /// </remarks>
81          [MethodImpl(MethodImplOptions.AggressiveInlining)]
82          public static void DeleteAll<TLink>(this ILinks<TLink> links)
83          {
84              var equalityComparer = EqualityComparer<TLink>.Default;
85              var comparer = Comparer<TLink>.Default;
86              for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
                ↪  Arithmetic.Decrement(i))
87              {
88                  links.Delete(i);
89                  if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
90                  {
91                      i = links.Count();
92                  }
93              }
```

```
94          }
95
96          [MethodImpl(MethodImplOptions.AggressiveInlining)]
97          public static TLink First<TLink>(this ILinks<TLink> links)
98          {
99              TLink firstLink = default;
100             var equalityComparer = EqualityComparer<TLink>.Default;
101             if (equalityComparer.Equals(links.Count(), default))
102             {
103                 throw new InvalidOperationException("В хранилище нет связей.");
104             }
105             links.Each(links.Constants.Any, links.Constants.Any, link =>
106             {
107                 firstLink = link[links.Constants.IndexPart];
108                 return links.Constants.Break;
109             });
110             if (equalityComparer.Equals(firstLink, default))
111             {
112                 throw new InvalidOperationException("В процессе поиска по хранилищу не было
                    ↪  найдено связей.");
113             }
114             return firstLink;
115         }
116
117         #region Paths
118
119         /// <remarks>
120         /// TODO: Как так? Как то что ниже может быть корректно?
121         /// Скорее всего практически не применимо
122         /// Предполагалось, что можно было конвертировать формируемый в проходе через
            ↪  SequenceWalker
123         /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
124         /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
125         /// </remarks>
126         [MethodImpl(MethodImplOptions.AggressiveInlining)]
127         public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
            ↪  path)
128         {
129             var current = path[0];
130             //EnsureLinkExists(current, "path");
131             if (!links.Exists(current))
132             {
133                 return false;
134             }
135             var equalityComparer = EqualityComparer<TLink>.Default;
136             var constants = links.Constants;
137             for (var i = 1; i < path.Length; i++)
138             {
139                 var next = path[i];
140                 var values = links.GetLink(current);
141                 var source = values[constants.SourcePart];
142                 var target = values[constants.TargetPart];
143                 if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
                    ↪  next))
144                 {
145                     //throw new InvalidOperationException(string.Format("Невозможно выбрать
                        ↪  путь, так как и Source и Target совпадают с элементом пути {0}.", next));
146                     return false;
147                 }
148                 if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
                    ↪  target))
149                 {
150                     //throw new InvalidOperationException(string.Format("Невозможно продолжить
                        ↪  путь через элемент пути {0}", next));
151                     return false;
152                 }
153                 current = next;
154             }
155             return true;
156         }
157
158         /// <remarks>
159         /// Может потребовать дополнительного стека для PathElement's при использовании
            ↪  SequenceWalker.
160         /// </remarks>
161         [MethodImpl(MethodImplOptions.AggressiveInlining)]
162         public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink root, params int[]
            ↪  path)
```

```
163         {
164             links.EnsureLinkExists(root, "root");
165             var currentLink = root;
166             for (var i = 0; i < path.Length; i++)
167             {
168                 currentLink = links.GetLink(currentLink)[path[i]];
169             }
170             return currentLink;
171         }
172 
173         [MethodImpl(MethodImplOptions.AggressiveInlining)]
174         public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
            ↪ links, TLink root, ulong size, ulong index)
175         {
176             var constants = links.Constants;
177             var source = constants.SourcePart;
178             var target = constants.TargetPart;
179             if (!Platform.Numbers.Math.IsPowerOfTwo(size))
180             {
181                 throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
                    ↪  than powers of two are not supported.");
182             }
183             var path = new BitArray(BitConverter.GetBytes(index));
184             var length = Bit.GetLowestPosition(size);
185             links.EnsureLinkExists(root, "root");
186             var currentLink = root;
187             for (var i = length - 1; i >= 0; i--)
188             {
189                 currentLink = links.GetLink(currentLink)[path[i] ? target : source];
190             }
191             return currentLink;
192         }
193 
194         #endregion
195 
196         /// <summary>
197         /// Возвращает индекс указанной связи.
198         /// </summary>
199         /// <param name="links">Хранилище связей.</param>
200         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪ содержимого.</param>
201         /// <returns>Индекс начальной связи для указанной связи.</returns>
202         [MethodImpl(MethodImplOptions.AggressiveInlining)]
203         public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪ link[links.Constants.IndexPart];
204 
205         /// <summary>
206         /// Возвращает индекс начальной (Source) связи для указанной связи.
207         /// </summary>
208         /// <param name="links">Хранилище связей.</param>
209         /// <param name="link">Индекс связи.</param>
210         /// <returns>Индекс начальной связи для указанной связи.</returns>
211         [MethodImpl(MethodImplOptions.AggressiveInlining)]
212         public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪ links.GetLink(link)[links.Constants.SourcePart];
213 
214         /// <summary>
215         /// Возвращает индекс начальной (Source) связи для указанной связи.
216         /// </summary>
217         /// <param name="links">Хранилище связей.</param>
218         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪ содержимого.</param>
219         /// <returns>Индекс начальной связи для указанной связи.</returns>
220         [MethodImpl(MethodImplOptions.AggressiveInlining)]
221         public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪ link[links.Constants.SourcePart];
222 
223         /// <summary>
224         /// Возвращает индекс конечной (Target) связи для указанной связи.
225         /// </summary>
226         /// <param name="links">Хранилище связей.</param>
227         /// <param name="link">Индекс связи.</param>
228         /// <returns>Индекс конечной связи для указанной связи.</returns>
229         [MethodImpl(MethodImplOptions.AggressiveInlining)]
230         public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪ links.GetLink(link)[links.Constants.TargetPart];
231 
232         /// <summary>
```

```
233         /// Возвращает индекс конечной (Target) связи для указанной связи.
234         /// </summary>
235         /// <param name="links">Хранилище связей.</param>
236         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪ содержимого.</param>
237         /// <returns>Индекс конечной связи для указанной связи.</returns>
238         [MethodImpl(MethodImplOptions.AggressiveInlining)]
239         public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪ link[links.Constants.TargetPart];
240
241         /// <summary>
242         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
243         /// </summary>
244         /// <param name="links">Хранилище связей.</param>
245         /// <param name="handler">Обработчик каждой подходящей связи.</param>
246         /// <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>
247         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
248         [MethodImpl(MethodImplOptions.AggressiveInlining)]
249         public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
            ↪ handler, params TLink[] restrictions)
250             => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
                ↪ links.Constants.Continue);
251
252         /// <summary>
253         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
254         /// </summary>
255         /// <param name="links">Хранилище связей.</param>
256         /// <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>
257         /// <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>
258         /// <param name="handler">Обработчик каждой подходящей связи.</param>
259         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
260         [MethodImpl(MethodImplOptions.AggressiveInlining)]
261         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
            ↪ Func<TLink, bool> handler)
262         {
263             var constants = links.Constants;
264             return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
                ↪ constants.Break, constants.Any, source, target);
265         }
266
267         /// <summary>
268         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
269         /// </summary>
270         /// <param name="links">Хранилище связей.</param>
271         /// <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>
272         /// <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>
273         /// <param name="handler">Обработчик каждой подходящей связи.</param>
274         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
275         [MethodImpl(MethodImplOptions.AggressiveInlining)]
276         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
            ↪ Func<IList<TLink>, TLink> handler) => links.Each(handler, links.Constants.Any,
            ↪ source, target);
277
278         [MethodImpl(MethodImplOptions.AggressiveInlining)]
279         public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ restrictions)
280         {
281             var arraySize = CheckedConverter<TLink,
                ↪ long>.Default.Convert(links.Count(restrictions));
282             if (arraySize > 0)
```

```
283             {
284                 var array = new IList<TLink>[arraySize];
285                 var filler = new ArrayFiller<IList<TLink>, TLink>(array,
                    ↪ links.Constants.Continue);
286                 links.Each(filler.AddAndReturnConstant, restrictions);
287                 return array;
288             }
289             else
290             {
291                 return Array.Empty<IList<TLink>>();
292             }
293         }
294 
295         [MethodImpl(MethodImplOptions.AggressiveInlining)]
296         public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ restrictions)
297         {
298             var arraySize = CheckedConverter<TLink,
                ↪ long>.Default.Convert(links.Count(restrictions));
299             if (arraySize > 0)
300             {
301                 var array = new TLink[arraySize];
302                 var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
303                 links.Each(filler.AddFirstAndReturnConstant, restrictions);
304                 return array;
305             }
306             else
307             {
308                 return Array.Empty<TLink>();
309             }
310         }
311 
312         /// <summary>
313         /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
            ↪ в хранилище связей.
314         /// </summary>
315         /// <param name="links">Хранилище связей.</param>
316         /// <param name="source">Начало связи.</param>
317         /// <param name="target">Конец связи.</param>
318         /// <returns>Значение, определяющее существует ли связь.</returns>
319         [MethodImpl(MethodImplOptions.AggressiveInlining)]
320         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;
321 
322         #region Ensure
323         // TODO: May be move to EnsureExtensions or make it both there and here
324 
325         [MethodImpl(MethodImplOptions.AggressiveInlining)]
326         public static void EnsureLinkExists<TLink>(this ILinks<TLink> links, IList<TLink>
            ↪ restrictions)
327         {
328             for (var i = 0; i < restrictions.Count; i++)
329             {
330                 if (!links.Exists(restrictions[i]))
331                 {
332                     throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                        ↪ $"sequence[{i}]");
333                 }
334             }
335         }
336 
337         [MethodImpl(MethodImplOptions.AggressiveInlining)]
338         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
            ↪ reference, string argumentName)
339         {
340             if (links.Constants.IsInternalReference(reference) && !links.Exists(reference))
341             {
342                 throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
343             }
344         }
345 
346         [MethodImpl(MethodImplOptions.AggressiveInlining)]
347         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
            ↪ IList<TLink> restrictions, string argumentName)
348         {
349             for (int i = 0; i < restrictions.Count; i++)
350             {
```

```
351                 links.EnsureInnerReferenceExists(restrictions[i], argumentName);
352             }
353         }
354 
355         [MethodImpl(MethodImplOptions.AggressiveInlining)]
356         public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
            ↪ restrictions)
357         {
358             var equalityComparer = EqualityComparer<TLink>.Default;
359             var any = links.Constants.Any;
360             for (var i = 0; i < restrictions.Count; i++)
361             {
362                 if (!equalityComparer.Equals(restrictions[i], any) &&
                    ↪ !links.Exists(restrictions[i]))
363                 {
364                     throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                        ↪ $"sequence[{i}]");
365                 }
366             }
367         }
368 
369         [MethodImpl(MethodImplOptions.AggressiveInlining)]
370         public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
            ↪ string argumentName)
371         {
372             var equalityComparer = EqualityComparer<TLink>.Default;
373             if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
374             {
375                 throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
376             }
377         }
378 
379         [MethodImpl(MethodImplOptions.AggressiveInlining)]
380         public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
            ↪ link, string argumentName)
381         {
382             var equalityComparer = EqualityComparer<TLink>.Default;
383             if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
384             {
385                 throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
386             }
387         }
388 
389         /// <param name="links">Хранилище связей.</param>
390         [MethodImpl(MethodImplOptions.AggressiveInlining)]
391         public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
            ↪ TLink target)
392         {
393             if (links.Exists(source, target))
394             {
395                 throw new LinkWithSameValueAlreadyExistsException();
396             }
397         }
398 
399         /// <param name="links">Хранилище связей.</param>
400         [MethodImpl(MethodImplOptions.AggressiveInlining)]
401         public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
402         {
403             if (links.HasUsages(link))
404             {
405                 throw new ArgumentLinkHasDependenciesException<TLink>(link);
406             }
407         }
408 
409         /// <param name="links">Хранилище связей.</param>
410         [MethodImpl(MethodImplOptions.AggressiveInlining)]
411         public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ addresses) => links.EnsureCreated(links.Create, addresses);
412 
413         /// <param name="links">Хранилище связей.</param>
414         [MethodImpl(MethodImplOptions.AggressiveInlining)]
415         public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
            ↪ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
416 
417         /// <param name="links">Хранилище связей.</param>
418         [MethodImpl(MethodImplOptions.AggressiveInlining)]
419         public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
            ↪ params TLink[] addresses)
```

```
420         {
421             var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
422             var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
423             var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
                ↪ !links.Exists(x)));
424             if (nonExistentAddresses.Count > 0)
425             {
426                 var max = nonExistentAddresses.Max();
427                 max = uInt64ToAddressConverter.Convert(System.Math.Min(addressToUInt64Converter.
                    ↪ Convert(max),
                    ↪ addressToUInt64Converter.Convert(links.Constants.InternalReferencesRange.Max
                    ↪ imum)));
428                 var createdLinks = new List<TLink>();
429                 var equalityComparer = EqualityComparer<TLink>.Default;
430                 TLink createdLink = creator();
431                 while (!equalityComparer.Equals(createdLink, max))
432                 {
433                     createdLinks.Add(createdLink);
434                 }
435                 for (var i = 0; i < createdLinks.Count; i++)
436                 {
437                     if (!nonExistentAddresses.Contains(createdLinks[i]))
438                     {
439                         links.Delete(createdLinks[i]);
440                     }
441                 }
442             }
443         }
444 
445         #endregion
446 
447         /// <param name="links">Хранилище связей.</param>
448         [MethodImpl(MethodImplOptions.AggressiveInlining)]
449         public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
450         {
451             var constants = links.Constants;
452             var values = links.GetLink(link);
453             TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link,
                ↪ constants.Any));
454             var equalityComparer = EqualityComparer<TLink>.Default;
455             if (equalityComparer.Equals(values[constants.SourcePart], link))
456             {
457                 usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
458             }
459             TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
                ↪ link));
460             if (equalityComparer.Equals(values[constants.TargetPart], link))
461             {
462                 usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
463             }
464             return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
465         }
466 
467         /// <param name="links">Хранилище связей.</param>
468         [MethodImpl(MethodImplOptions.AggressiveInlining)]
469         public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪ Comparer<TLink>.Default.Compare(links.CountUsages(link), default) > 0;
470 
471         /// <param name="links">Хранилище связей.</param>
472         [MethodImpl(MethodImplOptions.AggressiveInlining)]
473         public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
            ↪ TLink target)
474         {
475             var constants = links.Constants;
476             var values = links.GetLink(link);
477             var equalityComparer = EqualityComparer<TLink>.Default;
478             return equalityComparer.Equals(values[constants.SourcePart], source) &&
                ↪ equalityComparer.Equals(values[constants.TargetPart], target);
479         }
480 
481         /// <summary>
482         /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
483         /// </summary>
484         /// <param name="links">Хранилище связей.</param>
485         /// <param name="source">Индекс связи, которая является началом для искомой
            ↪ связи.</param>
486         /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
```

```
487         /// <returns>Индекс искомой связи с указанными Source (началом) и Target
            ↪  (концом).</returns>
488         [MethodImpl(MethodImplOptions.AggressiveInlining)]
489         public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪  target)
490         {
491             var contants = links.Constants;
492             var setter = new Setter<TLink, TLink>(contants.Continue, contants.Break, default);
493             links.Each(setter.SetFirstAndReturnFalse, contants.Any, source, target);
494             return setter.Result;
495         }
496
497         /// <param name="links">Хранилище связей.</param>
498         [MethodImpl(MethodImplOptions.AggressiveInlining)]
499         public static TLink Create<TLink>(this ILinks<TLink> links) => links.Create(null);
500
501         /// <param name="links">Хранилище связей.</param>
502         [MethodImpl(MethodImplOptions.AggressiveInlining)]
503         public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
504         {
505             var link = links.Create();
506             return links.Update(link, link, link);
507         }
508
509         /// <param name="links">Хранилище связей.</param>
510         [MethodImpl(MethodImplOptions.AggressiveInlining)]
511         public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪  target) => links.Update(links.Create(), source, target);
512
513         /// <summary>
514         /// Обновляет связь с указанными началом (Source) и концом (Target)
515         /// на связь с указанными началом (NewSource) и концом (NewTarget).
516         /// </summary>
517         /// <param name="links">Хранилище связей.</param>
518         /// <param name="link">Индекс обновляемой связи.</param>
519         /// <param name="newSource">Индекс связи, которая является началом связи, на которую
            ↪  выполняется обновление.</param>
520         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
            ↪  выполняется обновление.</param>
521         /// <returns>Индекс обновлённой связи.</returns>
522         [MethodImpl(MethodImplOptions.AggressiveInlining)]
523         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));
524
525         /// <summary>
526         /// Обновляет связь с указанными началом (Source) и концом (Target)
527         /// на связь с указанными началом (NewSource) и концом (NewTarget).
528         /// </summary>
529         /// <param name="links">Хранилище связей.</param>
530         /// <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>
531         /// <returns>Индекс обновлённой связи.</returns>
532         [MethodImpl(MethodImplOptions.AggressiveInlining)]
533         public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
534         {
535             if (restrictions.Length == 2)
536             {
537                 return links.MergeAndDelete(restrictions[0], restrictions[1]);
538             }
539             if (restrictions.Length == 4)
540             {
541                 return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
                    ↪  restrictions[2], restrictions[3]);
542             }
543             else
544             {
545                 return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
546             }
547         }
548
549         [MethodImpl(MethodImplOptions.AggressiveInlining)]
550         public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
            ↪  links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
551         {
552             var equalityComparer = EqualityComparer<TLink>.Default;
```

```
553             var constants = links.Constants;
554             var restrictionsIndex = restrictions[constants.IndexPart];
555             var substitutionIndex = substitution[constants.IndexPart];
556             if (equalityComparer.Equals(substitutionIndex, default))
557             {
558                 substitutionIndex = restrictionsIndex;
559             }
560             var source = substitution[constants.SourcePart];
561             var target = substitution[constants.TargetPart];
562             source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
563             target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
564             return new Link<TLink>(substitutionIndex, source, target);
565         }
566 
567         /// <summary>
568         /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
            ↪ с указанными Source (началом) и Target (концом).
569         /// </summary>
570         /// <param name="links">Хранилище связей.</param>
571         /// <param name="source">Индекс связи, которая является началом на создаваемой
            ↪ связи.</param>
572         /// <param name="target">Индекс связи, которая является концом для создаваемой
            ↪ связи.</param>
573         /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
574         [MethodImpl(MethodImplOptions.AggressiveInlining)]
575         public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target)
576         {
577             var link = links.SearchOrDefault(source, target);
578             if (EqualityComparer<TLink>.Default.Equals(link, default))
579             {
580                 link = links.CreateAndUpdate(source, target);
581             }
582             return link;
583         }
584 
585         /// <summary>
586         /// Обновляет связь с указанными началом (Source) и концом (Target)
587         /// на связь с указанными началом (NewSource) и концом (NewTarget).
588         /// </summary>
589         /// <param name="links">Хранилище связей.</param>
590         /// <param name="source">Индекс связи, которая является началом обновляемой
            ↪ связи.</param>
591         /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
592         /// <param name="newSource">Индекс связи, которая является началом связи, на которую
            ↪ выполняется обновление.</param>
593         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
            ↪ выполняется обновление.</param>
594         /// <returns>Индекс обновлённой связи.</returns>
595         [MethodImpl(MethodImplOptions.AggressiveInlining)]
596         public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
            ↪ TLink target, TLink newSource, TLink newTarget)
597         {
598             var equalityComparer = EqualityComparer<TLink>.Default;
599             var link = links.SearchOrDefault(source, target);
600             if (equalityComparer.Equals(link, default))
601             {
602                 return links.CreateAndUpdate(newSource, newTarget);
603             }
604             if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
                ↪ target))
605             {
606                 return link;
607             }
608             return links.Update(link, newSource, newTarget);
609         }
610 
611         /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
612         /// <param name="links">Хранилище связей.</param>
613         /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
614         /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
615         [MethodImpl(MethodImplOptions.AggressiveInlining)]
616         public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target)
617         {
618             var link = links.SearchOrDefault(source, target);
619             if (!EqualityComparer<TLink>.Default.Equals(link, default))
```

```
            {
                links.Delete(link);
                return link;
            }
            return default;
        }

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

        /// <remarks>Before execution of this method ensure that deleted link is detached (all values - source and target are reset to null) or it might enter into infinite recursion.</remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
        {
            var anyConstant = links.Constants.Any;
            var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
            links.DeleteByQuery(usagesAsSourceQuery);
            var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
            links.DeleteByQuery(usagesAsTargetQuery);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
        {
            var count = CheckedConverter<TLink, long>.Default.Convert(links.Count(query));
            if (count > 0)
            {
                var queryResult = new TLink[count];
                var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult, links.Constants.Continue);
                links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
                for (var i = count - 1; i >= 0; i--)
                {
                    links.Delete(queryResult[i]);
                }
            }
        }

        // TODO: Move to Platform.Data
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
        {
            var nullConstant = links.Constants.Null;
            var equalityComparer = EqualityComparer<TLink>.Default;
            var link = links.GetLink(linkIndex);
            for (int i = 1; i < link.Count; i++)
            {
                if (!equalityComparer.Equals(link[i], nullConstant))
                {
                    return false;
                }
            }
            return true;
        }

        // TODO: Create a universal version of this method in Platform.Data (with using of for loop)
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
        {
            var nullConstant = links.Constants.Null;
            var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
            links.Update(updateRequest);
        }

        // TODO: Create a universal version of this method in Platform.Data (with using of for loop)
```

```
693         [MethodImpl(MethodImplOptions.AggressiveInlining)]
694         public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
695         {
696             if (!links.AreValuesReset(linkIndex))
697             {
698                 links.ResetValues(linkIndex);
699             }
700         }
701 
702         /// <summary>
703         /// Merging two usages graphs, all children of old link moved to be children of new link
            ↪  or deleted.
704         /// </summary>
705         [MethodImpl(MethodImplOptions.AggressiveInlining)]
706         public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
            ↪  TLink newLinkIndex)
707         {
708             var addressToInt64Converter = CheckedConverter<TLink, long>.Default;
709             var equalityComparer = EqualityComparer<TLink>.Default;
710             if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
711             {
712                 var constants = links.Constants;
713                 var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
                    ↪  constants.Any);
714                 var usagesAsSourceCount =
                    ↪  addressToInt64Converter.Convert(links.Count(usagesAsSourceQuery));
715                 var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
                    ↪  oldLinkIndex);
716                 var usagesAsTargetCount =
                    ↪  addressToInt64Converter.Convert(links.Count(usagesAsTargetQuery));
717                 var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
                    ↪  usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
718                 if (!isStandalonePoint)
719                 {
720                     var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
721                     if (totalUsages > 0)
722                     {
723                         var usages = ArrayPool.Allocate<TLink>(totalUsages);
724                         var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
                            ↪  links.Constants.Continue);
725                         var i = 0L;
726                         if (usagesAsSourceCount > 0)
727                         {
728                             links.Each(usagesFiller.AddFirstAndReturnConstant,
                                ↪  usagesAsSourceQuery);
729                             for (; i < usagesAsSourceCount; i++)
730                             {
731                                 var usage = usages[i];
732                                 if (!equalityComparer.Equals(usage, oldLinkIndex))
733                                 {
734                                     links.Update(usage, newLinkIndex, links.GetTarget(usage));
735                                 }
736                             }
737                         }
738                         if (usagesAsTargetCount > 0)
739                         {
740                             links.Each(usagesFiller.AddFirstAndReturnConstant,
                                ↪  usagesAsTargetQuery);
741                             for (; i < usages.Length; i++)
742                             {
743                                 var usage = usages[i];
744                                 if (!equalityComparer.Equals(usage, oldLinkIndex))
745                                 {
746                                     links.Update(usage, links.GetSource(usage), newLinkIndex);
747                                 }
748                             }
749                         }
750                         ArrayPool.Free(usages);
751                     }
752                 }
753             }
754             return newLinkIndex;
755         }
756 
757         /// <summary>
758         /// Replace one link with another (replaced link is deleted, children are updated or
            ↪  deleted).
759         /// </summary>
```

```
760         [MethodImpl(MethodImplOptions.AggressiveInlining)]
761         public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
            ↪ TLink newLinkIndex)
762         {
763             var equalityComparer = EqualityComparer<TLink>.Default;
764             if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
765             {
766                 links.MergeUsages(oldLinkIndex, newLinkIndex);
767                 links.Delete(oldLinkIndex);
768             }
769             return newLinkIndex;
770         }
771 
772         [MethodImpl(MethodImplOptions.AggressiveInlining)]
773         public static ILinks<TLink>
            ↪ DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
774         {
775             links = new LinksCascadeUsagesResolver<TLink>(links);
776             links = new NonNullContentsLinkDeletionResolver<TLink>(links);
777             links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
778             return links;
779         }
780 
781         [MethodImpl(MethodImplOptions.AggressiveInlining)]
782         public static string Format<TLink>(this ILinks<TLink> links, IList<TLink> link)
783         {
784             var constants = links.Constants;
785             return $"({link[constants.IndexPart]}: {link[constants.SourcePart]}
                ↪ {link[constants.TargetPart]})";
786         }
787 
788         [MethodImpl(MethodImplOptions.AggressiveInlining)]
789         public static string Format<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪ links.Format(links.GetLink(link));
790     }
791 }
```

1.20 ./csharp/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  }
```

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

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Incrementers;
4 
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6 
7  namespace Platform.Data.Doublets.Incrementers
8  {
9      public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪ EqualityComparer<TLink>.Default;
12 
13         private readonly TLink _frequencyMarker;
14         private readonly TLink _unaryOne;
15         private readonly IIncrementer<TLink> _unaryNumberIncrementer;
16 
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
           ↪ IIncrementer<TLink> unaryNumberIncrementer)
19             : base(links)
20         {
21             _frequencyMarker = frequencyMarker;
22             _unaryOne = unaryOne;
23             _unaryNumberIncrementer = unaryNumberIncrementer;
24         }
25 
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Increment(TLink frequency)
28         {
```

```
            var links = _links;
            if (_equalityComparer.Equals(frequency, default))
            {
                return links.GetOrCreate(_unaryOne, _frequencyMarker);
            }
            var incrementedSource =
            ↪  _unaryNumberIncrementer.Increment(links.GetSource(frequency));
            return links.GetOrCreate(incrementedSource, _frequencyMarker);
        }
    }
}
```

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

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

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

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

        private readonly TLink _unaryOne;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
        ↪  _unaryOne = unaryOne;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Increment(TLink unaryNumber)
        {
            var links = _links;
            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));
            }
        }
    }
}
```

1.23 ./csharp/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;

```

```
24          private const int Length = 3;
25
26          public readonly TLink Index;
27          public readonly TLink Source;
28          public readonly TLink Target;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public Link(params TLink[] values) => SetValues(values, out Index, out Source, out
            ↪  Target);
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public Link(IList<TLink> values) => SetValues(values, out Index, out Source, out Target);
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          public Link(object other)
38          {
39              if (other is Link<TLink> otherLink)
40              {
41                  SetValues(ref otherLink, out Index, out Source, out Target);
42              }
43              else if(other is IList<TLink> otherList)
44              {
45                  SetValues(otherList, out Index, out Source, out Target);
46              }
47              else
48              {
49                  throw new NotSupportedException();
50              }
51          }
52
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          public Link(ref Link<TLink> other) => SetValues(ref other, out Index, out Source, out
            ↪  Target);
55
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          public Link(TLink index, TLink source, TLink target)
58          {
59              Index = index;
60              Source = source;
61              Target = target;
62          }
63
64          [MethodImpl(MethodImplOptions.AggressiveInlining)]
65          private static void SetValues(ref Link<TLink> other, out TLink index, out TLink source,
            ↪  out TLink target)
66          {
67              index = other.Index;
68              source = other.Source;
69              target = other.Target;
70          }
71
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          private static void SetValues(IList<TLink> values, out TLink index, out TLink source,
            ↪  out TLink target)
74          {
75              switch (values.Count)
76              {
77                  case 3:
78                      index = values[0];
79                      source = values[1];
80                      target = values[2];
81                      break;
82                  case 2:
83                      index = values[0];
84                      source = values[1];
85                      target = default;
86                      break;
87                  case 1:
88                      index = values[0];
89                      source = default;
90                      target = default;
91                      break;
92                  default:
93                      index = default;
94                      source = default;
95                      target = default;
96                      break;
97              }
98          }
99
```

```
100         [MethodImpl(MethodImplOptions.AggressiveInlining)]
101         public override int GetHashCode() => (Index, Source, Target).GetHashCode();
102 
103         [MethodImpl(MethodImplOptions.AggressiveInlining)]
104         public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
105                             && _equalityComparer.Equals(Source, _constants.Null)
106                             && _equalityComparer.Equals(Target, _constants.Null);
107 
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         public override bool Equals(object other) => other is Link<TLink> &&
            ↪  Equals((Link<TLink>)other);
110 
111         [MethodImpl(MethodImplOptions.AggressiveInlining)]
112         public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
113                                               && _equalityComparer.Equals(Source, other.Source)
114                                               && _equalityComparer.Equals(Target, other.Target);
115 
116         [MethodImpl(MethodImplOptions.AggressiveInlining)]
117         public static string ToString(TLink index, TLink source, TLink target) => $"({index}:
            ↪  {source}->{target})";
118 
119         [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
134         {
135             [MethodImpl(MethodImplOptions.AggressiveInlining)]
136             get => Length;
137         }
138 
139         public bool IsReadOnly
140         {
141             [MethodImpl(MethodImplOptions.AggressiveInlining)]
142             get => true;
143         }
144 
145         public TLink this[int index]
146         {
147             [MethodImpl(MethodImplOptions.AggressiveInlining)]
148             get
149             {
150                 Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
                    ↪  nameof(index));
151                 if (index == _constants.IndexPart)
152                 {
153                     return Index;
154                 }
155                 if (index == _constants.SourcePart)
156                 {
157                     return Source;
158                 }
159                 if (index == _constants.TargetPart)
160                 {
161                     return Target;
162                 }
163                 throw new NotSupportedException(); // Impossible path due to
                    ↪  Ensure.ArgumentInRange
164             }
165             [MethodImpl(MethodImplOptions.AggressiveInlining)]
166             set => throw new NotSupportedException();
167         }
168 
169         [MethodImpl(MethodImplOptions.AggressiveInlining)]
170         IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
171 
172         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
173         public IEnumerator<TLink> GetEnumerator()
174         {
175             yield return Index;
176             yield return Source;
177             yield return Target;
178         }
179 
180         [MethodImpl(MethodImplOptions.AggressiveInlining)]
181         public void Add(TLink item) => throw new NotSupportedException();
182 
183         [MethodImpl(MethodImplOptions.AggressiveInlining)]
184         public void Clear() => throw new NotSupportedException();
185 
186         [MethodImpl(MethodImplOptions.AggressiveInlining)]
187         public bool Contains(TLink item) => IndexOf(item) >= 0;
188 
189         [MethodImpl(MethodImplOptions.AggressiveInlining)]
190         public void CopyTo(TLink[] array, int arrayIndex)
191         {
192             Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
193             Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
                ↪  nameof(arrayIndex));
194             if (arrayIndex + Length > array.Length)
195             {
196                 throw new InvalidOperationException();
197             }
198             array[arrayIndex++] = Index;
199             array[arrayIndex++] = Source;
200             array[arrayIndex] = Target;
201         }
202 
203         [MethodImpl(MethodImplOptions.AggressiveInlining)]
204         public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
205 
206         [MethodImpl(MethodImplOptions.AggressiveInlining)]
207         public int IndexOf(TLink item)
208         {
209             if (_equalityComparer.Equals(Index, item))
210             {
211                 return _constants.IndexPart;
212             }
213             if (_equalityComparer.Equals(Source, item))
214             {
215                 return _constants.SourcePart;
216             }
217             if (_equalityComparer.Equals(Target, item))
218             {
219                 return _constants.TargetPart;
220             }
221             return -1;
222         }
223 
224         [MethodImpl(MethodImplOptions.AggressiveInlining)]
225         public void Insert(int index, TLink item) => throw new NotSupportedException();
226 
227         [MethodImpl(MethodImplOptions.AggressiveInlining)]
228         public void RemoveAt(int index) => throw new NotSupportedException();
229 
230         [MethodImpl(MethodImplOptions.AggressiveInlining)]
231         public static bool operator ==(Link<TLink> left, Link<TLink> right) =>
            ↪  left.Equals(right);
232 
233         [MethodImpl(MethodImplOptions.AggressiveInlining)]
234         public static bool operator !=(Link<TLink> left, Link<TLink> right) => !(left == right);
235 
236         #endregion
237     }
238 }
```

1.24 ./csharp/Platform.Data.Doublets/LinkExtensions.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
6  {
7      public static class LinkExtensions
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        public static bool IsFullPoint<TLink>(this Link<TLink> link) =>
        ↪ Point<TLink>.IsFullPoint(link);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
        ↪ Point<TLink>.IsPartialPoint(link);
    }
}
```

1.25 ./csharp/Platform.Data.Doublets/LinksOperatorBase.cs

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets
{
    public abstract class LinksOperatorBase<TLink>
    {
        protected readonly ILinks<TLink> _links;

        public ILinks<TLink> Links
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _links;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksOperatorBase(ILinks<TLink> links) => _links = links;
    }
}
```

1.26 ./csharp/Platform.Data.Doublets/Memory/ILinksListMethods.cs

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory
{
    public interface ILinksListMethods<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        void Detach(TLink freeLink);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        void AttachAsFirst(TLink link);
    }
}
```

1.27 ./csharp/Platform.Data.Doublets/Memory/ILinksTreeMethods.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.Memory
{
    public interface ILinksTreeMethods<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TLink CountUsages(TLink root);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TLink Search(TLink source, TLink target);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TLink EachUsage(TLink root, Func<IList<TLink>, TLink> handler);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        void Detach(ref TLink root, TLink linkIndex);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        void Attach(ref TLink root, TLink linkIndex);
    }
}
```

1.28 ./csharp/Platform.Data.Doublets/Memory/LinksHeader.cs

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

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

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

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

        public TLink AllocatedLinks;
        public TLink ReservedLinks;
        public TLink FreeLinks;
        public TLink FirstFreeLink;
        public TLink RootAsSource;
        public TLink RootAsTarget;
        public TLink LastFreeLink;
        public TLink Reserved8;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is LinksHeader<TLink> linksHeader ? Equals(linksHeader) : false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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(RootAsSource, other.RootAsSource)
            && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
            && _equalityComparer.Equals(LastFreeLink, other.LastFreeLink)
            && _equalityComparer.Equals(Reserved8, other.Reserved8);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => (AllocatedLinks, ReservedLinks, FreeLinks, FirstFreeLink, RootAsSource, RootAsTarget, LastFreeLink, Reserved8).GetHashCode();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator ==(LinksHeader<TLink> left, LinksHeader<TLink> right) => left.Equals(right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator !=(LinksHeader<TLink> left, LinksHeader<TLink> right) => !(left == right);
    }
}
```

1.29 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSizeBalancedTreeMethodsBase.cs

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

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe abstract class ExternalLinksSizeBalancedTreeMethodsBase<TLink> : SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter = UncheckedConverter<TLink, long>.Default;

        protected readonly TLink Break;
        protected readonly TLink Continue;
        protected readonly byte* LinksDataParts;
        protected readonly byte* LinksIndexParts;
        protected readonly byte* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected ExternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants,
        ↪  byte* linksDataParts, byte* linksIndexParts, byte* header)
        {
            LinksDataParts = linksDataParts;
            LinksIndexParts = linksIndexParts;
            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 RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
        ↪  AsRef<RawLinkDataPart<TLink>>(LinksDataParts + RawLinkDataPart<TLink>.SizeInBytes *
        ↪  _addressToInt64Converter.Convert(link));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
        ↪  ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
        ↪  RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
        {
            ref var link = ref GetLinkDataPartReference(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 GetLinkDataPartReference(first);
            ref var secondLink = ref GetLinkDataPartReference(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 GetLinkDataPartReference(first);
            ref var secondLink = ref GetLinkDataPartReference(second);
            return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
            ↪  secondLink.Source, secondLink.Target);
        }

        public TLink this[TLink index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            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>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public TLink Search(TLink source, TLink target)
    {
        var root = GetTreeRoot();
        while (!EqualToZero(root))
        {
            ref var rootLink = ref GetLinkDataPartReference(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
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    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);
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void PrintNodeValue(TLink node, StringBuilder sb)
        {
            ref var link = ref GetLinkDataPartReference(node);
            sb.Append(' ');
            sb.Append(link.Source);
            sb.Append('-');
            sb.Append('>');
            sb.Append(link.Target);
        }
    }
}
```

1.30 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSourcesSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
```

```
    public unsafe class ExternalLinksSourcesSizeBalancedTreeMethods<TLink> :
    ↪ ExternalLinksSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ExternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants,
        ↪ byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
        ↪ linksDataParts, linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) =>
        ↪ GetLinkDataPartReference(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 GetLinkIndexPartReference(node);
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

1.31 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe class ExternalLinksTargetsSizeBalancedTreeMethods<TLink> :
    ↪ ExternalLinksSizeBalancedTreeMethodsBase<TLink>
    {
```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ExternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants,
        ↪  byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
        ↪  linksDataParts, linksIndexParts, header) { }

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) =>
        ↪  GetLinkDataPartReference(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 GetLinkIndexPartReference(node);
            link.LeftAsTarget = Zero;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }
    }
}
```

1.32 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSizeBalancedTreeMethodsBase.cs

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

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
```

```
13      public unsafe abstract class InternalLinksSizeBalancedTreeMethodsBase<TLink> :
        ↪ SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
14      {
15          private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            ↪ UncheckedConverter<TLink, long>.Default;
16
17          protected readonly TLink Break;
18          protected readonly TLink Continue;
19          protected readonly byte* LinksDataParts;
20          protected readonly byte* LinksIndexParts;
21          protected readonly byte* Header;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected InternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants,
            ↪ byte* linksDataParts, byte* linksIndexParts, byte* header)
25          {
26              LinksDataParts = linksDataParts;
27              LinksIndexParts = linksIndexParts;
28              Header = header;
29              Break = constants.Break;
30              Continue = constants.Continue;
31          }
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected abstract TLink GetTreeRoot(TLink link);
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected abstract TLink GetBasePartValue(TLink link);
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected abstract TLink GetKeyPartValue(TLink link);
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
            ↪ AsRef<RawLinkDataPart<TLink>>(LinksDataParts + RawLinkDataPart<TLink>.SizeInBytes *
            ↪ _addressToInt64Converter.Convert(link));
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
            ↪ ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
            ↪ RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link));
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
            ↪ LessThan(GetKeyPartValue(first), GetKeyPartValue(second));
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
            ↪ GreaterThan(GetKeyPartValue(first), GetKeyPartValue(second));
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
56          {
57              ref var link = ref GetLinkDataPartReference(linkIndex);
58              return new Link<TLink>(linkIndex, link.Source, link.Target);
59          }
60
61          public TLink this[TLink link, TLink index]
62          {
63              [MethodImpl(MethodImplOptions.AggressiveInlining)]
64              get
65              {
66                  var root = GetTreeRoot(link);
67                  if (GreaterOrEqualThan(index, GetSize(root)))
68                  {
69                      return Zero;
70                  }
71                  while (!EqualToZero(root))
72                  {
73                      var left = GetLeftOrDefault(root);
74                      var leftSize = GetSizeOrZero(left);
75                      if (LessThan(index, leftSize))
76                      {
77                          root = left;
78                          continue;
79                      }
80                      if (AreEqual(index, leftSize))
81                      {
82                          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>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public abstract TLink Search(TLink source, TLink target);

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    protected TLink SearchCore(TLink root, TLink key)
    {
        while (!EqualToZero(root))
        {
            var rootKey = GetKeyPartValue(root);
            if (LessThan(key, rootKey)) // node.Key < root.Key
            {
                root = GetLeftOrDefault(root);
            }
            else if (GreaterThan(key, rootKey)) // node.Key > root.Key
            {
                root = GetRightOrDefault(root);
            }
            else // node.Key == root.Key
            {
                return root;
            }
        }
        return Zero;
    }

    // TODO: Return indices range instead of references count
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public TLink CountUsages(TLink link) => GetSizeOrZero(GetTreeRoot(link));

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

    // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use low-level MSIL stack.
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
    {
        var @continue = Continue;
        if (EqualToZero(link))
        {
            return @continue;
        }
        var @break = Break;
        if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
        {
            return @break;
        }
        if (AreEqual(handler(GetLinkValues(link)), @break))
        {
            return @break;
        }
        if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
        {
            return @break;
        }
        return @continue;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    protected override void PrintNodeValue(TLink node, StringBuilder sb)
    {
        ref var link = ref GetLinkDataPartReference(node);
```

```
            sb.Append(' ');
            sb.Append(link.Source);
            sb.Append('-');
            sb.Append('>');
            sb.Append(link.Target);
        }
    }
}
```

1.33 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe class InternalLinksSourcesSizeBalancedTreeMethods<TLink> :
    ↪ InternalLinksSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants,
        ↪ byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
        ↪ linksDataParts, linksIndexParts, header) { }

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot(TLink link) =>
        ↪ GetLinkIndexPartReference(link).RootAsSource;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetKeyPartValue(TLink link) =>
        ↪ GetLinkDataPartReference(link).Target;

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

        public override TLink Search(TLink source, TLink target) =>
        ↪ SearchCore(GetTreeRoot(source), target);
```

```
    }
}
```

1.34 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe class InternalLinksTargetsSizeBalancedTreeMethods<TLink> : InternalLinksSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public InternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants, linksDataParts, linksIndexParts, header) { }

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot(TLink link) => GetLinkIndexPartReference(link).RootAsTarget;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkIndexPartReference(node);
            link.LeftAsTarget = Zero;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }

        public override TLink Search(TLink source, TLink target) => SearchCore(GetTreeRoot(target), source);
    }
}
```

1.35 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinks.cs

```
using System;
using System.Runtime.CompilerServices;
```

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

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public unsafe class SplitMemoryLinks<TLink> : SplitMemoryLinksBase<TLink>
    {
        private readonly Func<ILinksTreeMethods<TLink>> _createInternalSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createExternalSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createInternalTargetTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createExternalTargetTreeMethods;
        private byte* _header;
        private byte* _linksDataParts;
        private byte* _linksIndexParts;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪  indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪  indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
        ↪  memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪  indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
        ↪  base(dataMemory, indexMemory, memoryReservationStep, constants)
        {
            _createInternalSourceTreeMethods = () => new
            ↪  InternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
            ↪  _linksIndexParts, _header);
            _createExternalSourceTreeMethods = () => new
            ↪  ExternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
            ↪  _linksIndexParts, _header);
            _createInternalTargetTreeMethods = () => new
            ↪  InternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
            ↪  _linksIndexParts, _header);
            _createExternalTargetTreeMethods = () => new
            ↪  ExternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
            ↪  _linksIndexParts, _header);
            Init(dataMemory, indexMemory, memoryReservationStep);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetPointers(IResizableDirectMemory dataMemory,
        ↪  IResizableDirectMemory indexMemory)
        {
            _linksDataParts = (byte*)dataMemory.Pointer;
            _linksIndexParts = (byte*)indexMemory.Pointer;
            _header = _linksIndexParts;
            InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
            ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
            InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
            ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
            UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_linksDataParts, _header);
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
        ↪  => ref AsRef<RawLinkDataPart<TLink>>(_linksDataParts + LinkDataPartSizeInBytes *
        ↪  ConvertToInt64(linkIndex));

```

```
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
            ↪  linkIndex) => ref AsRef<RawLinkIndexPart<TLink>>(_linksIndexParts +
            ↪  LinkIndexPartSizeInBytes * ConvertToInt64(linkIndex));
67      }
68  }
```

1.36 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinksBase.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.Converters;
7   using Platform.Numbers;
8   using Platform.Memory;
9   using Platform.Data.Exceptions;
10
11  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13  namespace Platform.Data.Doublets.Memory.Split.Generic
14  {
15      public abstract class SplitMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16      {
17          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
18          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
19          private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            ↪  UncheckedConverter<TLink, long>.Default;
20          private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
            ↪  UncheckedConverter<long, TLink>.Default;
21
22          private static readonly TLink _zero = default;
23          private static readonly TLink _one = Arithmetic.Increment(_zero);
24
25          /// <summary>Возвращает размер одной связи в байтах.</summary>
26          /// <remarks>
27          /// Используется только во вне класса, не рекомедуется использовать внутри.
28          /// Так как во вне не обязательно будет доступен unsafe C#.
29          /// </remarks>
30          public static readonly long LinkDataPartSizeInBytes = RawLinkDataPart<TLink>.SizeInBytes;
31
32          public static readonly long LinkIndexPartSizeInBytes =
            ↪  RawLinkIndexPart<TLink>.SizeInBytes;
33
34          public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
35
36          public static readonly long DefaultLinksSizeStep = 1 * 1024 * 1024;
37
38          protected readonly IResizableDirectMemory _dataMemory;
39          protected readonly IResizableDirectMemory _indexMemory;
40          protected readonly long _dataMemoryReservationStepInBytes;
41          protected readonly long _indexMemoryReservationStepInBytes;
42
43          protected ILinksTreeMethods<TLink> InternalSourcesTreeMethods;
44          protected ILinksTreeMethods<TLink> ExternalSourcesTreeMethods;
45          protected ILinksTreeMethods<TLink> InternalTargetsTreeMethods;
46          protected ILinksTreeMethods<TLink> ExternalTargetsTreeMethods;
47          // 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
48          protected ILinksListMethods<TLink> UnusedLinksListMethods;
49
50          /// <summary>
51          /// Возвращает общее число связей находящихся в хранилище.
52          /// </summary>
53          protected virtual TLink Total
54          {
55              [MethodImpl(MethodImplOptions.AggressiveInlining)]
56              get
57              {
58                  ref var header = ref GetHeaderReference();
59                  return Subtract(header.AllocatedLinks, header.FreeLinks);
60              }
61          }
62
63          public virtual LinksConstants<TLink> Constants
64          {
65              [MethodImpl(MethodImplOptions.AggressiveInlining)]
66              get;
67          }
68
```

```
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants)
71          {
72              _dataMemory = dataMemory;
73              _indexMemory = indexMemory;
74              _dataMemoryReservationStepInBytes = memoryReservationStep * LinkDataPartSizeInBytes;
75              _indexMemoryReservationStepInBytes = memoryReservationStep *
                ↪ LinkIndexPartSizeInBytes;
76              Constants = constants;
77          }
78
79          [MethodImpl(MethodImplOptions.AggressiveInlining)]
80          protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
            ↪ memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }
81
82          [MethodImpl(MethodImplOptions.AggressiveInlining)]
83          protected virtual void Init(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↪ indexMemory, long memoryReservationStep)
84          {
85              if (dataMemory.ReservedCapacity < memoryReservationStep)
86              {
87                  dataMemory.ReservedCapacity = memoryReservationStep;
88              }
89              if (indexMemory.ReservedCapacity < memoryReservationStep)
90              {
91                  indexMemory.ReservedCapacity = memoryReservationStep;
92              }
93              SetPointers(dataMemory, indexMemory);
94              ref var header = ref GetHeaderReference();
95              // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
96              // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
97              dataMemory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) *
                ↪ LinkDataPartSizeInBytes + LinkDataPartSizeInBytes; // First link is read only
                ↪ zero link.
98              indexMemory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) *
                ↪ LinkIndexPartSizeInBytes + LinkHeaderSizeInBytes;
99              // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
100             // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
101             header.ReservedLinks = ConvertToAddress((dataMemory.ReservedCapacity -
                ↪ LinkDataPartSizeInBytes) / LinkDataPartSizeInBytes);
102         }
103
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         public virtual TLink Count(IList<TLink> restrictions)
106         {
107             // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
108             if (restrictions.Count == 0)
109             {
110                 return Total;
111             }
112             var constants = Constants;
113             var any = constants.Any;
114             var index = restrictions[constants.IndexPart];
115             if (restrictions.Count == 1)
116             {
117                 if (AreEqual(index, any))
118                 {
119                     return Total;
120                 }
121                 return Exists(index) ? GetOne() : GetZero();
122             }
123             if (restrictions.Count == 2)
124             {
125                 var value = restrictions[1];
126                 if (AreEqual(index, any))
127                 {
128                     if (AreEqual(value, any))
129                     {
130                         return Total; // Any - как отсутствие ограничения
131                     }
132                     var externalReferencesRange = constants.ExternalReferencesRange;
133                     if (externalReferencesRange.HasValue &&
                        ↪ externalReferencesRange.Value.Contains(value))
134                     {
135                         return Add(ExternalSourcesTreeMethods.CountUsages(value),
                            ↪ ExternalTargetsTreeMethods.CountUsages(value));
```

```
136                     }
137                     else
138                     {
139                         return Add(InternalSourcesTreeMethods.CountUsages(value),
                            ↪  InternalTargetsTreeMethods.CountUsages(value));
140                     }
141                 }
142                 else
143                 {
144                     if (!Exists(index))
145                     {
146                         return GetZero();
147                     }
148                     if (AreEqual(value, any))
149                     {
150                         return GetOne();
151                     }
152                     ref var storedLinkValue = ref GetLinkDataPartReference(index);
153                     if (AreEqual(storedLinkValue.Source, value) ||
                        ↪  AreEqual(storedLinkValue.Target, value))
154                     {
155                         return GetOne();
156                     }
157                     return GetZero();
158                 }
159             }
160             if (restrictions.Count == 3)
161             {
162                 var externalReferencesRange = constants.ExternalReferencesRange;
163                 var source = restrictions[constants.SourcePart];
164                 var target = restrictions[constants.TargetPart];
165                 if (AreEqual(index, any))
166                 {
167                     if (AreEqual(source, any) && AreEqual(target, any))
168                     {
169                         return Total;
170                     }
171                     else if (AreEqual(source, any))
172                     {
173                         if (externalReferencesRange.HasValue &&
                            ↪  externalReferencesRange.Value.Contains(target))
174                         {
175                             return ExternalTargetsTreeMethods.CountUsages(target);
176                         }
177                         else
178                         {
179                             return InternalTargetsTreeMethods.CountUsages(target);
180                         }
181                     }
182                     else if (AreEqual(target, any))
183                     {
184                         if (externalReferencesRange.HasValue &&
                            ↪  externalReferencesRange.Value.Contains(source))
185                         {
186                             return ExternalSourcesTreeMethods.CountUsages(source);
187                         }
188                         else
189                         {
190                             return InternalSourcesTreeMethods.CountUsages(source);
191                         }
192                     }
193                     else //if(source != Any && target != Any)
194                     {
195                         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
196                         TLink link;
197                         if (externalReferencesRange.HasValue)
198                         {
199                             if (externalReferencesRange.Value.Contains(source) &&
                                ↪  externalReferencesRange.Value.Contains(target))
200                             {
201                                 link = ExternalSourcesTreeMethods.Search(source, target);
202                             }
203                             else if (externalReferencesRange.Value.Contains(source))
204                             {
205                                 link = InternalTargetsTreeMethods.Search(source, target);
206                             }
207                             else if (externalReferencesRange.Value.Contains(target))
208                             {
```

```
                                link = InternalSourcesTreeMethods.Search(source, target);
                            }
                            else
                            {
                                if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                                ↪  InternalTargetsTreeMethods.CountUsages(target)))
                                {
                                    link = InternalTargetsTreeMethods.Search(source, target);
                                }
                                else
                                {
                                    link = InternalSourcesTreeMethods.Search(source, target);
                                }
                            }
                        }
                        else
                        {
                            if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                            ↪  InternalTargetsTreeMethods.CountUsages(target)))
                            {
                                link = InternalTargetsTreeMethods.Search(source, target);
                            }
                            else
                            {
                                link = InternalSourcesTreeMethods.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 GetLinkDataPartReference(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)
            {
```

```
282                     for (var link = GetOne(); LessOrEqualThan(link,
                        ↪  GetHeaderReference().AllocatedLinks); link = Increment(link))
283                     {
284                         if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
285                         {
286                             return @break;
287                         }
288                     }
289                     return @break;
290                 }
291                 var @continue = constants.Continue;
292                 var any = constants.Any;
293                 var index = restrictions[constants.IndexPart];
294                 if (restrictions.Count == 1)
295                 {
296                     if (AreEqual(index, any))
297                     {
298                         return Each(handler, Array.Empty<TLink>());
299                     }
300                     if (!Exists(index))
301                     {
302                         return @continue;
303                     }
304                     return handler(GetLinkStruct(index));
305                 }
306                 if (restrictions.Count == 2)
307                 {
308                     var value = restrictions[1];
309                     if (AreEqual(index, any))
310                     {
311                         if (AreEqual(value, any))
312                         {
313                             return Each(handler, Array.Empty<TLink>());
314                         }
315                         if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
316                         {
317                             return @break;
318                         }
319                         return Each(handler, new Link<TLink>(index, any, value));
320                     }
321                     else
322                     {
323                         if (!Exists(index))
324                         {
325                             return @continue;
326                         }
327                         if (AreEqual(value, any))
328                         {
329                             return handler(GetLinkStruct(index));
330                         }
331                         ref var storedLinkValue = ref GetLinkDataPartReference(index);
332                         if (AreEqual(storedLinkValue.Source, value) ||
333                             AreEqual(storedLinkValue.Target, value))
334                         {
335                             return handler(GetLinkStruct(index));
336                         }
337                         return @continue;
338                     }
339                 }
340                 if (restrictions.Count == 3)
341                 {
342                     var externalReferencesRange = constants.ExternalReferencesRange;
343                     var source = restrictions[constants.SourcePart];
344                     var target = restrictions[constants.TargetPart];
345                     if (AreEqual(index, any))
346                     {
347                         if (AreEqual(source, any) && AreEqual(target, any))
348                         {
349                             return Each(handler, Array.Empty<TLink>());
350                         }
351                         else if (AreEqual(source, any))
352                         {
353                             if (externalReferencesRange.HasValue &&
                                ↪  externalReferencesRange.Value.Contains(target))
354                             {
355                                 return ExternalTargetsTreeMethods.EachUsage(target, handler);
356                             }
357                             else
```

```
                    {
                        return InternalTargetsTreeMethods.EachUsage(target, handler);
                    }
                }
                else if (AreEqual(target, any))
                {
                    if (externalReferencesRange.HasValue &&
                    ↪  externalReferencesRange.Value.Contains(source))
                    {
                        return ExternalSourcesTreeMethods.EachUsage(source, handler);
                    }
                    else
                    {
                        return InternalSourcesTreeMethods.EachUsage(source, handler);
                    }
                }
                else //if(source != Any && target != Any)
                {
                    TLink link;
                    if (externalReferencesRange.HasValue)
                    {
                        if (externalReferencesRange.Value.Contains(source) &&
                        ↪  externalReferencesRange.Value.Contains(target))
                        {
                            link = ExternalSourcesTreeMethods.Search(source, target);
                        }
                        else if (externalReferencesRange.Value.Contains(source))
                        {
                            link = InternalTargetsTreeMethods.Search(source, target);
                        }
                        else if (externalReferencesRange.Value.Contains(target))
                        {
                            link = InternalSourcesTreeMethods.Search(source, target);
                        }
                        else
                        {
                            if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                            ↪  InternalTargetsTreeMethods.CountUsages(target)))
                            {
                                link = InternalTargetsTreeMethods.Search(source, target);
                            }
                            else
                            {
                                link = InternalSourcesTreeMethods.Search(source, target);
                            }
                        }
                    }
                    else
                    {
                        if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                        ↪  InternalTargetsTreeMethods.CountUsages(target)))
                        {
                            link = InternalTargetsTreeMethods.Search(source, target);
                        }
                        else
                        {
                            link = InternalSourcesTreeMethods.Search(source, target);
                        }
                    }
                    return AreEqual(link, constants.Null) ? @continue :
                    ↪  handler(GetLinkStruct(link));
                }
            }
            else
            {
                if (!Exists(index))
                {
                    return @continue;
                }
                if (AreEqual(source, any) && AreEqual(target, any))
                {
                    return handler(GetLinkStruct(index));
                }
                ref var storedLinkValue = ref GetLinkDataPartReference(index);
                if (!AreEqual(source, any) && !AreEqual(target, any))
                {
                    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 externalReferencesRange = constants.ExternalReferencesRange;
        var linkIndex = restrictions[constants.IndexPart];
        ref var link = ref GetLinkDataPartReference(linkIndex);
        var source = link.Source;
        var target = link.Target;
        ref var header = ref GetHeaderReference();
        ref var rootAsSource = ref header.RootAsSource;
        ref var rootAsTarget = ref header.RootAsTarget;
        // Будет корректно работать только в том случае, если пространство выделенной связи
        ↪  предварительно заполнено нулями
        if (!AreEqual(source, @null))
        {
            if (externalReferencesRange.HasValue &&
            ↪  externalReferencesRange.Value.Contains(source))
            {
                ExternalSourcesTreeMethods.Detach(ref rootAsSource, linkIndex);
            }
            else
            {
                InternalSourcesTreeMethods.Detach(ref
                ↪  GetLinkIndexPartReference(source).RootAsSource, linkIndex);
            }
        }
        if (!AreEqual(target, @null))
        {
            if (externalReferencesRange.HasValue &&
            ↪  externalReferencesRange.Value.Contains(target))
            {
                ExternalTargetsTreeMethods.Detach(ref rootAsTarget, linkIndex);
            }
            else
            {
                InternalTargetsTreeMethods.Detach(ref
                ↪  GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
            }
        }
        source = link.Source = substitution[constants.SourcePart];
        target = link.Target = substitution[constants.TargetPart];
        if (!AreEqual(source, @null))
        {
            if (externalReferencesRange.HasValue &&
            ↪  externalReferencesRange.Value.Contains(source))
            {
```

```
501                     ExternalSourcesTreeMethods.Attach(ref rootAsSource, linkIndex);
502                 }
503                 else
504                 {
505                     InternalSourcesTreeMethods.Attach(ref
                        ↪  GetLinkIndexPartReference(source).RootAsSource, linkIndex);
506                 }
507             }
508             if (!AreEqual(target, @null))
509             {
510                 if (externalReferencesRange.HasValue &&
                    ↪  externalReferencesRange.Value.Contains(target))
511                 {
512                     ExternalTargetsTreeMethods.Attach(ref rootAsTarget, linkIndex);
513                 }
514                 else
515                 {
516                     InternalTargetsTreeMethods.Attach(ref
                        ↪  GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
517                 }
518             }
519             return linkIndex;
520         }
521 
522         /// <remarks>
523         /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
            ↪  пространство
524         /// </remarks>
525         [MethodImpl(MethodImplOptions.AggressiveInlining)]
526         public virtual TLink Create(IList<TLink> restrictions)
527         {
528             ref var header = ref GetHeaderReference();
529             var freeLink = header.FirstFreeLink;
530             if (!AreEqual(freeLink, Constants.Null))
531             {
532                 UnusedLinksListMethods.Detach(freeLink);
533             }
534             else
535             {
536                 var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
537                 if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
538                 {
539                     throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
540                 }
541                 if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
542                 {
543                     _dataMemory.ReservedCapacity += _dataMemoryReservationStepInBytes;
544                     _indexMemory.ReservedCapacity += _indexMemoryReservationStepInBytes;
545                     SetPointers(_dataMemory, _indexMemory);
546                     header = ref GetHeaderReference();
547                     header.ReservedLinks = ConvertToAddress(_dataMemory.ReservedCapacity /
                        ↪  LinkDataPartSizeInBytes);
548                 }
549                 header.AllocatedLinks = Increment(header.AllocatedLinks);
550                 _dataMemory.UsedCapacity += LinkDataPartSizeInBytes;
551                 _indexMemory.UsedCapacity += LinkIndexPartSizeInBytes;
552                 freeLink = header.AllocatedLinks;
553             }
554             return freeLink;
555         }
556 
557         [MethodImpl(MethodImplOptions.AggressiveInlining)]
558         public virtual void Delete(IList<TLink> restrictions)
559         {
560             ref var header = ref GetHeaderReference();
561             var link = restrictions[Constants.IndexPart];
562             if (LessThan(link, header.AllocatedLinks))
563             {
564                 UnusedLinksListMethods.AttachAsFirst(link);
565             }
566             else if (AreEqual(link, header.AllocatedLinks))
567             {
568                 header.AllocatedLinks = Decrement(header.AllocatedLinks);
569                 _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
570                 _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
571                 // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                    ↪  пока не дойдём до первой существующей связи
572                 // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
```

```
573                 while (GreaterThan(header.AllocatedLinks, GetZero()) &&
                    ↪  IsUnusedLink(header.AllocatedLinks))
574                 {
575                     UnusedLinksListMethods.Detach(header.AllocatedLinks);
576                     header.AllocatedLinks = Decrement(header.AllocatedLinks);
577                     _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
578                     _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
579                 }
580             }
581         }
582
583         [MethodImpl(MethodImplOptions.AggressiveInlining)]
584         public IList<TLink> GetLinkStruct(TLink linkIndex)
585         {
586             ref var link = ref GetLinkDataPartReference(linkIndex);
587             return new Link<TLink>(linkIndex, link.Source, link.Target);
588         }
589
590         /// <remarks>
591         /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
                ↪  адрес реально поменялся
592         ///
593         /// Указатель this.links может быть в том же месте,
594         /// так как 0-я связь не используется и имеет такой же размер как Header,
595         /// поэтому header размещается в том же месте, что и 0-я связь
596         /// </remarks>
597         [MethodImpl(MethodImplOptions.AggressiveInlining)]
598         protected abstract void SetPointers(IResizableDirectMemory dataMemory,
                ↪  IResizableDirectMemory indexMemory);
599
600         [MethodImpl(MethodImplOptions.AggressiveInlining)]
601         protected virtual void ResetPointers()
602         {
603             InternalSourcesTreeMethods = null;
604             ExternalSourcesTreeMethods = null;
605             InternalTargetsTreeMethods = null;
606             ExternalTargetsTreeMethods = null;
607             UnusedLinksListMethods = null;
608         }
609
610         [MethodImpl(MethodImplOptions.AggressiveInlining)]
611         protected abstract ref LinksHeader<TLink> GetHeaderReference();
612
613         [MethodImpl(MethodImplOptions.AggressiveInlining)]
614         protected abstract ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex);
615
616         [MethodImpl(MethodImplOptions.AggressiveInlining)]
617         protected abstract ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
                ↪  linkIndex);
618
619         [MethodImpl(MethodImplOptions.AggressiveInlining)]
620         protected virtual bool Exists(TLink link)
621             => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
622             && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
623             && !IsUnusedLink(link);
624
625         [MethodImpl(MethodImplOptions.AggressiveInlining)]
626         protected virtual bool IsUnusedLink(TLink linkIndex)
627         {
628             if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
                    ↪  is not needed
629             {
630                 // TODO: Reduce access to memory in different location (should be enough to use
                        ↪  just linkIndexPart)
631                 ref var linkDataPart = ref GetLinkDataPartReference(linkIndex);
632                 ref var linkIndexPart = ref GetLinkIndexPartReference(linkIndex);
633                 return AreEqual(linkIndexPart.SizeAsSource, default) &&
                        ↪  !AreEqual(linkDataPart.Source, default);
634             }
635             else
636             {
637                 return true;
638             }
639         }
640
641         [MethodImpl(MethodImplOptions.AggressiveInlining)]
642         protected virtual TLink GetOne() => _one;
643
644         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
645         protected virtual TLink GetZero() => default;
646
647         [MethodImpl(MethodImplOptions.AggressiveInlining)]
648         protected virtual bool AreEqual(TLink first, TLink second) =>
            ↪ _equalityComparer.Equals(first, second);
649
650         [MethodImpl(MethodImplOptions.AggressiveInlining)]
651         protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
            ↪ second) < 0;
652
653         [MethodImpl(MethodImplOptions.AggressiveInlining)]
654         protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
            ↪ _comparer.Compare(first, second) <= 0;
655
656         [MethodImpl(MethodImplOptions.AggressiveInlining)]
657         protected virtual bool GreaterThan(TLink first, TLink second) =>
            ↪ _comparer.Compare(first, second) > 0;
658
659         [MethodImpl(MethodImplOptions.AggressiveInlining)]
660         protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
            ↪ _comparer.Compare(first, second) >= 0;
661
662         [MethodImpl(MethodImplOptions.AggressiveInlining)]
663         protected virtual long ConvertToInt64(TLink value) =>
            ↪ _addressToInt64Converter.Convert(value);
664
665         [MethodImpl(MethodImplOptions.AggressiveInlining)]
666         protected virtual TLink ConvertToAddress(long value) =>
            ↪ _int64ToAddressConverter.Convert(value);
667
668         [MethodImpl(MethodImplOptions.AggressiveInlining)]
669         protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
            ↪ second);
670
671         [MethodImpl(MethodImplOptions.AggressiveInlining)]
672         protected virtual TLink Subtract(TLink first, TLink second) =>
            ↪ Arithmetic<TLink>.Subtract(first, second);
673
674         [MethodImpl(MethodImplOptions.AggressiveInlining)]
675         protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
676
677         [MethodImpl(MethodImplOptions.AggressiveInlining)]
678         protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
679
680         #region Disposable
681
682         protected override bool AllowMultipleDisposeCalls
683         {
684             [MethodImpl(MethodImplOptions.AggressiveInlining)]
685             get => true;
686         }
687
688         [MethodImpl(MethodImplOptions.AggressiveInlining)]
689         protected override void Dispose(bool manual, bool wasDisposed)
690         {
691             if (!wasDisposed)
692             {
693                 ResetPointers();
694                 _dataMemory.DisposeIfPossible();
695                 _indexMemory.DisposeIfPossible();
696             }
697         }
698
699         #endregion
700     }
701 }
```

1.37 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/UnusedLinksListMethods.cs

```
1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Methods.Lists;
3  using Platform.Converters;
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.Memory.Split.Generic
9  {
10     public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
       ↪ ILinksListMethods<TLink>
```

```
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
        ↪  UncheckedConverter<TLink, long>.Default;

        private readonly byte* _links;
        private readonly byte* _header;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
        ↪  AsRef<RawLinkDataPart<TLink>>(_links + RawLinkDataPart<TLink>.SizeInBytes *
        ↪  _addressToInt64Converter.Convert(link));

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

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

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

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

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

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

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

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

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

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

1.38 ./csharp/Platform.Data.Doublets/Memory/Split/RawLinkDataPart.cs

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

        public static readonly long SizeInBytes = Structure<RawLinkDataPart<TLink>>.Size;

        public TLink Source;
        public TLink Target;
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is RawLinkDataPart<TLink> link ?
        ↪  Equals(link) : false;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator ==(RawLinkDataPart<TLink> left, RawLinkDataPart<TLink>
        ↪  right) => left.Equals(right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator !=(RawLinkDataPart<TLink> left, RawLinkDataPart<TLink>
        ↪  right) => !(left == right);
    }
}
```

1.39 ./csharp/Platform.Data.Doublets/Memory/Split/RawLinkIndexPart.cs

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

        public static readonly long SizeInBytes = Structure<RawLinkIndexPart<TLink>>.Size;

        public TLink RootAsSource;
        public TLink LeftAsSource;
        public TLink RightAsSource;
        public TLink SizeAsSource;
        public TLink RootAsTarget;
        public TLink LeftAsTarget;
        public TLink RightAsTarget;
        public TLink SizeAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is RawLinkIndexPart<TLink> link ?
        ↪  Equals(link) : false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(RawLinkIndexPart<TLink> other)
            => _equalityComparer.Equals(RootAsSource, other.RootAsSource)
            && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
            && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
            && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
            && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
            && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
            && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
            && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => (RootAsSource, LeftAsSource, RightAsSource,
        ↪  SizeAsSource, RootAsTarget, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator ==(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
        ↪  right) => left.Equals(right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator !=(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
        ↪  right) => !(left == right);
    }
}
```

1.40 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksAvlBalancedTreeMethodsBase.cs

```
using System;
using System.Text;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Methods.Trees;
using Platform.Converters;
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.Memory.United.Generic
{
    public unsafe abstract class LinksAvlBalancedTreeMethodsBase<TLink> : SizedAndThreadedAVLBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter = UncheckedConverter<TLink, long>.Default;
        private static readonly UncheckedConverter<TLink, int> _addressToInt32Converter = UncheckedConverter<TLink, int>.Default;
        private static readonly UncheckedConverter<bool, TLink> _boolToAddressConverter = UncheckedConverter<bool, TLink>.Default;
        private static readonly UncheckedConverter<TLink, bool> _addressToBoolConverter = UncheckedConverter<TLink, bool>.Default;
        private static readonly UncheckedConverter<int, TLink> _int32ToAddressConverter = UncheckedConverter<int, TLink>.Default;

        protected readonly TLink Break;
        protected readonly TLink Continue;
        protected readonly byte* Links;
        protected readonly byte* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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 * _addressToInt64Converter.Convert(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 _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 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,
                ↪  _boolToAddressConverter.Convert(value), 4, 1);
                storedValue = modified;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GetRightIsChildValue(TLink value)
        {
            unchecked
            {
                return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 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,
                ↪  _boolToAddressConverter.Convert(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 = _addressToInt32Converter.Convert(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 = _int32ToAddressConverter.Convert((byte)value >> 5 & 4 | value & 3);
                var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
                storedValue = modified;
            }
        }

        public TLink this[TLink index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    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);
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    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;
        }

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

1.41 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSizeBalancedTreeMethodsBase.cs

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

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

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe abstract class LinksSizeBalancedTreeMethodsBase<TLink> : SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter = UncheckedConverter<TLink, long>.Default;

        protected readonly TLink Break;
        protected readonly TLink Continue;
        protected readonly byte* Links;
        protected readonly byte* Header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links, byte* header)
        {
            Links = links;
            Header = header;
            Break = constants.Break;
            Continue = constants.Continue;
        }

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref AsRef<RawLink<TLink>>(Links + RawLink<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
50          protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
51          {
52              ref var link = ref GetLinkReference(linkIndex);
53              return new Link<TLink>(linkIndex, link.Source, link.Target);
54          }
55
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
58          {
59              ref var firstLink = ref GetLinkReference(first);
60              ref var secondLink = ref GetLinkReference(second);
61              return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
62          }
63
64          [MethodImpl(MethodImplOptions.AggressiveInlining)]
65          protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
66          {
67              ref var firstLink = ref GetLinkReference(first);
68              ref var secondLink = ref GetLinkReference(second);
69              return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
70          }
71
72          public TLink this[TLink index]
73          {
74              [MethodImpl(MethodImplOptions.AggressiveInlining)]
75              get
76              {
77                  var root = GetTreeRoot();
78                  if (GreaterOrEqualThan(index, GetSize(root)))
79                  {
80                      return Zero;
81                  }
82                  while (!EqualToZero(root))
83                  {
84                      var left = GetLeftOrDefault(root);
85                      var leftSize = GetSizeOrZero(left);
86                      if (LessThan(index, leftSize))
87                      {
88                          root = left;
89                          continue;
90                      }
91                      if (AreEqual(index, leftSize))
92                      {
93                          return root;
94                      }
95                      root = GetRightOrDefault(root);
96                      index = Subtract(index, Increment(leftSize));
97                  }
98                  return Zero; // TODO: Impossible situation exception (only if tree structure
                    ↪  broken)
99              }
100         }
101
102         /// <summary>
103         /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
                ↪  (концом).
104         /// </summary>
105         /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
106         /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
107         /// <returns>Индекс искомой связи.</returns>
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         public TLink Search(TLink source, TLink target)
110         {
111             var root = GetTreeRoot();
112             while (!EqualToZero(root))
113             {
114                 ref var rootLink = ref GetLinkReference(root);
115                 var rootSource = rootLink.Source;
116                 var rootTarget = rootLink.Target;
117                 if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
                    ↪  node.Key < root.Key
118                 {
119                     root = GetLeftOrDefault(root);
120                 }
121                 else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
                    ↪  node.Key > root.Key
```

```
122                 {
123                     root = GetRightOrDefault(root);
124                 }
125                 else // node.Key == root.Key
126                 {
127                     return root;
128                 }
129             }
130             return Zero;
131         }
132 
133         // TODO: Return indices range instead of references count
134         [MethodImpl(MethodImplOptions.AggressiveInlining)]
135         public TLink CountUsages(TLink link)
136         {
137             var root = GetTreeRoot();
138             var total = GetSize(root);
139             var totalRightIgnore = Zero;
140             while (!EqualToZero(root))
141             {
142                 var @base = GetBasePartValue(root);
143                 if (LessOrEqualThan(@base, link))
144                 {
145                     root = GetRightOrDefault(root);
146                 }
147                 else
148                 {
149                     totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
150                     root = GetLeftOrDefault(root);
151                 }
152             }
153             root = GetTreeRoot();
154             var totalLeftIgnore = Zero;
155             while (!EqualToZero(root))
156             {
157                 var @base = GetBasePartValue(root);
158                 if (GreaterOrEqualThan(@base, link))
159                 {
160                     root = GetLeftOrDefault(root);
161                 }
162                 else
163                 {
164                     totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
165                     root = GetRightOrDefault(root);
166                 }
167             }
168             return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
169         }
170 
171         [MethodImpl(MethodImplOptions.AggressiveInlining)]
172         public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
            ↪ EachUsageCore(@base, GetTreeRoot(), handler);
173 
174         // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
            ↪ low-level MSIL stack.
175         [MethodImpl(MethodImplOptions.AggressiveInlining)]
176         private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
177         {
178             var @continue = Continue;
179             if (EqualToZero(link))
180             {
181                 return @continue;
182             }
183             var linkBasePart = GetBasePartValue(link);
184             var @break = Break;
185             if (GreaterThan(linkBasePart, @base))
186             {
187                 if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
188                 {
189                     return @break;
190                 }
191             }
192             else if (LessThan(linkBasePart, @base))
193             {
194                 if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
195                 {
196                     return @break;
197                 }
198             }
```

```
            else //if (linkBasePart == @base)
            {
                if (AreEqual(handler(GetLinkValues(link)), @break))
                {
                    return @break;
                }
                if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
                {
                    return @break;
                }
                if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
                {
                    return @break;
                }
            }
            return @continue;
        }

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

1.42 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesAvlBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe class LinksSourcesAvlBalancedTreeMethods<TLink> :
    ↪ LinksAvlBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksSourcesAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
        ↪ byte* header) : base(constants, links, header) { }

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

        [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;
        }
    }
}
```

1.43 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe class LinksSourcesSizeBalancedTreeMethods<TLink> :
    ↪  LinksSizeBalancedTreeMethodsBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
        ↪  byte* header) : base(constants, links, header) { }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected 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;

```

```
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected override void SetRight(TLink node, TLink right) =>
            ↪  GetLinkReference(node).RightAsSource = right;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsSource;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override void SetSize(TLink node, TLink size) =>
            ↪  GetLinkReference(node).SizeAsSource = size;
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
            ↪  AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget);
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
            ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
            ↪  AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget);
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override void ClearNode(TLink node)
50          {
51              ref var link = ref GetLinkReference(node);
52              link.LeftAsSource = Zero;
53              link.RightAsSource = Zero;
54              link.SizeAsSource = Zero;
55          }
56      }
57  }
```

1.44 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsAvlBalancedTreeMethods.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.Memory.United.Generic
6   {
7       public unsafe class LinksTargetsAvlBalancedTreeMethods<TLink> :
        ↪  LinksAvlBalancedTreeMethodsBase<TLink>
8       {
9           [MethodImpl(MethodImplOptions.AggressiveInlining)]
10          public LinksTargetsAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
            ↪  byte* header) : base(constants, links, header) { }
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          protected override ref TLink GetLeftReference(TLink node) => ref
            ↪  GetLinkReference(node).LeftAsTarget;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected override ref TLink GetRightReference(TLink node) => ref
            ↪  GetLinkReference(node).RightAsTarget;
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override void SetLeft(TLink node, TLink left) =>
            ↪  GetLinkReference(node).LeftAsTarget = left;
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          protected override void SetRight(TLink node, TLink right) =>
            ↪  GetLinkReference(node).RightAsTarget = right;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          protected override TLink GetSize(TLink node) =>
            ↪  GetSizeValue(GetLinkReference(node).SizeAsTarget);
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
34          protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
            ↪ GetLinkReference(node).SizeAsTarget, size);
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override bool GetLeftIsChild(TLink node) =>
            ↪ GetLeftIsChildValue(GetLinkReference(node).SizeAsTarget);
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override void SetLeftIsChild(TLink node, bool value) =>
            ↪ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override bool GetRightIsChild(TLink node) =>
            ↪ GetRightIsChildValue(GetLinkReference(node).SizeAsTarget);
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override void SetRightIsChild(TLink node, bool value) =>
            ↪ SetRightIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override sbyte GetBalance(TLink node) =>
            ↪ GetBalanceValue(GetLinkReference(node).SizeAsTarget);
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
            ↪ GetLinkReference(node).SizeAsTarget, value);
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
59
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
            ↪ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
            ↪ AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource);
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
            ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
            ↪ AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource);
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          protected override void ClearNode(TLink node)
68          {
69              ref var link = ref GetLinkReference(node);
70              link.LeftAsTarget = Zero;
71              link.RightAsTarget = Zero;
72              link.SizeAsTarget = Zero;
73          }
74      }
75  }
```

1.45 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsSizeBalancedTreeMethods.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.Memory.United.Generic
6   {
7       public unsafe class LinksTargetsSizeBalancedTreeMethods<TLink> :
        ↪ LinksSizeBalancedTreeMethodsBase<TLink>
8       {
9           [MethodImpl(MethodImplOptions.AggressiveInlining)]
10          public LinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
            ↪ byte* header) : base(constants, links, header) { }
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          protected override ref TLink GetLeftReference(TLink node) => ref
            ↪ GetLinkReference(node).LeftAsTarget;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected override ref TLink GetRightReference(TLink node) => ref
            ↪ GetLinkReference(node).RightAsTarget;
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          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().RootAsTarget;

        [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;
        }
    }
}
```

1.46 ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinks.cs

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

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

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe class UnitedMemoryLinks<TLink> : UnitedMemoryLinksBase<TLink>
    {
        private readonly Func<ILinksTreeMethods<TLink>> _createSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createTargetTreeMethods;
        private byte* _header;
        private byte* _links;

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

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

```

```
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          public UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
            ↪  DefaultLinksSizeStep) { }
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep) :
            ↪  this(memory, memoryReservationStep, Default<LinksConstants<TLink>>.Instance, true) {
            ↪  }
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep,
            ↪  LinksConstants<TLink> constants, bool useAvlBasedIndex) : base(memory,
            ↪  memoryReservationStep, constants)
37          {
38              if (useAvlBasedIndex)
39              {
40                  _createSourceTreeMethods = () => new
                    ↪  LinksSourcesAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
41                  _createTargetTreeMethods = () => new
                    ↪  LinksTargetsAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
42              }
43              else
44              {
45                  _createSourceTreeMethods = () => new
                    ↪  LinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
46                  _createTargetTreeMethods = () => new
                    ↪  LinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
47              }
48              Init(memory, memoryReservationStep);
49          }
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void SetPointers(IResizableDirectMemory memory)
53          {
54              _links = (byte*)memory.Pointer;
55              _header = _links;
56              SourcesTreeMethods = _createSourceTreeMethods();
57              TargetsTreeMethods = _createTargetTreeMethods();
58              UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_links, _header);
59          }
60
61          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          protected override void ResetPointers()
63          {
64              base.ResetPointers();
65              _links = null;
66              _header = null;
67          }
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          protected override ref LinksHeader<TLink> GetHeaderReference() => ref
            ↪  AsRef<LinksHeader<TLink>>(_header);
71
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          protected override ref RawLink<TLink> GetLinkReference(TLink linkIndex) => ref
            ↪  AsRef<RawLink<TLink>>(_links + LinkSizeInBytes * ConvertToInt64(linkIndex));
74      }
75  }
```

1.47 ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinksBase.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.Converters;
7   using Platform.Numbers;
8   using Platform.Memory;
9   using Platform.Data.Exceptions;
10
11  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13  namespace Platform.Data.Doublets.Memory.United.Generic
14  {
15      public abstract class UnitedMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16      {
17          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
18          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
```

```
19          private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            ↪  UncheckedConverter<TLink, long>.Default;
20          private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
            ↪  UncheckedConverter<long, TLink>.Default;
21
22          private static readonly TLink _zero = default;
23          private static readonly TLink _one = Arithmetic.Increment(_zero);
24
25          /// <summary>Возвращает размер одной связи в байтах.</summary>
26          /// <remarks>
27          /// Используется только во вне класса, не рекомедуется использовать внутри.
28          /// Так как во вне не обязательно будет доступен unsafe C#.
29          /// </remarks>
30          public static readonly long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;
31
32          public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
33
34          public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
35
36          protected readonly IResizableDirectMemory _memory;
37          protected readonly long _memoryReservationStep;
38
39          protected ILinksTreeMethods<TLink> TargetsTreeMethods;
40          protected ILinksTreeMethods<TLink> SourcesTreeMethods;
41          // 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
42          protected ILinksListMethods<TLink> UnusedLinksListMethods;
43
44          /// <summary>
45          /// Возвращает общее число связей находящихся в хранилище.
46          /// </summary>
47          protected virtual TLink Total
48          {
49              [MethodImpl(MethodImplOptions.AggressiveInlining)]
50              get
51              {
52                  ref var header = ref GetHeaderReference();
53                  return Subtract(header.AllocatedLinks, header.FreeLinks);
54              }
55          }
56
57          public virtual LinksConstants<TLink> Constants
58          {
59              [MethodImpl(MethodImplOptions.AggressiveInlining)]
60              get;
61          }
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
            ↪  memoryReservationStep, LinksConstants<TLink> constants)
65          {
66              _memory = memory;
67              _memoryReservationStep = memoryReservationStep;
68              Constants = constants;
69          }
70
71          [MethodImpl(MethodImplOptions.AggressiveInlining)]
72          protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
            ↪  memoryReservationStep) : this(memory, memoryReservationStep,
            ↪  Default<LinksConstants<TLink>>.Instance) { }
73
74          [MethodImpl(MethodImplOptions.AggressiveInlining)]
75          protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
76          {
77              if (memory.ReservedCapacity < memoryReservationStep)
78              {
79                  memory.ReservedCapacity = memoryReservationStep;
80              }
81              SetPointers(memory);
82              ref var header = ref GetHeaderReference();
83              // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
84              memory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) * LinkSizeInBytes +
                ↪  LinkHeaderSizeInBytes;
85              // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
86              header.ReservedLinks = ConvertToAddress((memory.ReservedCapacity -
                ↪  LinkHeaderSizeInBytes) / LinkSizeInBytes);
87          }
88
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
90          public virtual TLink Count(IList<TLink> restrictions)
91          {
92              // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
93              if (restrictions.Count == 0)
94              {
95                  return Total;
96              }
97              var constants = Constants;
98              var any = constants.Any;
99              var index = restrictions[constants.IndexPart];
100             if (restrictions.Count == 1)
101             {
102                 if (AreEqual(index, any))
103                 {
104                     return Total;
105                 }
106                 return Exists(index) ? GetOne() : GetZero();
107             }
108             if (restrictions.Count == 2)
109             {
110                 var value = restrictions[1];
111                 if (AreEqual(index, any))
112                 {
113                     if (AreEqual(value, any))
114                     {
115                         return Total; // Any - как отсутствие ограничения
116                     }
117                     return Add(SourcesTreeMethods.CountUsages(value),
                        ↪ TargetsTreeMethods.CountUsages(value));
118                 }
119                 else
120                 {
121                     if (!Exists(index))
122                     {
123                         return GetZero();
124                     }
125                     if (AreEqual(value, any))
126                     {
127                         return GetOne();
128                     }
129                     ref var storedLinkValue = ref GetLinkReference(index);
130                     if (AreEqual(storedLinkValue.Source, value) ||
                        ↪ AreEqual(storedLinkValue.Target, value))
131                     {
132                         return GetOne();
133                     }
134                     return GetZero();
135                 }
136             }
137             if (restrictions.Count == 3)
138             {
139                 var source = restrictions[constants.SourcePart];
140                 var target = restrictions[constants.TargetPart];
141                 if (AreEqual(index, any))
142                 {
143                     if (AreEqual(source, any) && AreEqual(target, any))
144                     {
145                         return Total;
146                     }
147                     else if (AreEqual(source, any))
148                     {
149                         return TargetsTreeMethods.CountUsages(target);
150                     }
151                     else if (AreEqual(target, any))
152                     {
153                         return SourcesTreeMethods.CountUsages(source);
154                     }
155                     else //if(source != Any && target != Any)
156                     {
157                         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
158                         var link = SourcesTreeMethods.Search(source, target);
159                         return AreEqual(link, constants.Null) ? GetZero() : GetOne();
160                     }
161                 }
162                 else
163                 {
164                     if (!Exists(index))
165                     {
```

```
                        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, Array.Empty<TLink>());
                }
                if (!Exists(index))
                {
                    return @continue;
                }
                return handler(GetLinkStruct(index));
            }
            if (restrictions.Count == 2)
            {
                var value = restrictions[1];
                if (AreEqual(index, any))
                {
                    if (AreEqual(value, any))
                    {
                        return Each(handler, Array.Empty<TLink>());
                    }
```

```
                    if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
                    {
                        return @break;
                    }
                    return Each(handler, new Link<TLink>(index, any, value));
                }
                else
                {
                    if (!Exists(index))
                    {
                        return @continue;
                    }
                    if (AreEqual(value, any))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    ref var storedLinkValue = ref GetLinkReference(index);
                    if (AreEqual(storedLinkValue.Source, value) ||
                        AreEqual(storedLinkValue.Target, value))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return @continue;
                }
            }
            if (restrictions.Count == 3)
            {
                var source = restrictions[constants.SourcePart];
                var target = restrictions[constants.TargetPart];
                if (AreEqual(index, any))
                {
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return Each(handler, Array.Empty<TLink>());
                    }
                    else if (AreEqual(source, any))
                    {
                        return TargetsTreeMethods.EachUsage(target, handler);
                    }
                    else if (AreEqual(target, any))
                    {
                        return SourcesTreeMethods.EachUsage(source, handler);
                    }
                    else //if(source != Any && target != Any)
                    {
                        var link = SourcesTreeMethods.Search(source, target);
                        return AreEqual(link, constants.Null) ? @continue :
                        ↪  handler(GetLinkStruct(link));
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return @continue;
                    }
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    ref var storedLinkValue = ref GetLinkReference(index);
                    if (!AreEqual(source, any) && !AreEqual(target, any))
                    {
                        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;
```

```
317                     }
318                     if (AreEqual(storedLinkValue.Source, value) ||
319                         AreEqual(storedLinkValue.Target, value))
320                     {
321                         return handler(GetLinkStruct(index));
322                     }
323                     return @continue;
324                 }
325             }
326             throw new NotSupportedException("Другие размеры и способы ограничений не
                ↪  поддерживаются.");
327         }
328 
329         /// <remarks>
330         /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
                ↪  в другом месте (но не в менеджере памяти, а в логике Links)
331         /// </remarks>
332         [MethodImpl(MethodImplOptions.AggressiveInlining)]
333         public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
334         {
335             var constants = Constants;
336             var @null = constants.Null;
337             var linkIndex = restrictions[constants.IndexPart];
338             ref var link = ref GetLinkReference(linkIndex);
339             ref var header = ref GetHeaderReference();
340             ref var firstAsSource = ref header.RootAsSource;
341             ref var firstAsTarget = ref header.RootAsTarget;
342             // Будет корректно работать только в том случае, если пространство выделенной связи
                ↪  предварительно заполнено нулями
343             if (!AreEqual(link.Source, @null))
344             {
345                 SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
346             }
347             if (!AreEqual(link.Target, @null))
348             {
349                 TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
350             }
351             link.Source = substitution[constants.SourcePart];
352             link.Target = substitution[constants.TargetPart];
353             if (!AreEqual(link.Source, @null))
354             {
355                 SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
356             }
357             if (!AreEqual(link.Target, @null))
358             {
359                 TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
360             }
361             return linkIndex;
362         }
363 
364         /// <remarks>
365         /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
                ↪  пространство
366         /// </remarks>
367         [MethodImpl(MethodImplOptions.AggressiveInlining)]
368         public virtual TLink Create(IList<TLink> restrictions)
369         {
370             ref var header = ref GetHeaderReference();
371             var freeLink = header.FirstFreeLink;
372             if (!AreEqual(freeLink, Constants.Null))
373             {
374                 UnusedLinksListMethods.Detach(freeLink);
375             }
376             else
377             {
378                 var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
379                 if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
380                 {
381                     throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
382                 }
383                 if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
384                 {
385                     _memory.ReservedCapacity += _memoryReservationStep;
386                     SetPointers(_memory);
387                     header = ref GetHeaderReference();
388                     header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
                        ↪  LinkSizeInBytes);
389                 }
```

```
                header.AllocatedLinks = Increment(header.AllocatedLinks);
                _memory.UsedCapacity += LinkSizeInBytes;
                freeLink = header.AllocatedLinks;
            }
            return freeLink;
        }

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

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

        /// <remarks>
        /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если адрес реально поменялся
        ///
        /// Указатель this.links может быть в том же месте,
        /// так как 0-я связь не используется и имеет такой же размер как Header,
        /// поэтому header размещается в том же месте, что и 0-я связь
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetPointers(IResizableDirectMemory memory);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void ResetPointers()
        {
            SourcesTreeMethods = null;
            TargetsTreeMethods = null;
            UnusedLinksListMethods = null;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract ref LinksHeader<TLink> GetHeaderReference();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract ref RawLink<TLink> GetLinkReference(TLink linkIndex);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool Exists(TLink link)
            => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
            && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
            && !IsUnusedLink(link);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool IsUnusedLink(TLink linkIndex)
        {
            if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check is not needed
            {
                ref var link = ref GetLinkReference(linkIndex);
                return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
```

```
            }
            else
            {
                return true;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink GetOne() => _one;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink GetZero() => default;

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual long ConvertToInt64(TLink value) =>
        ↪ _addressToInt64Converter.Convert(value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TLink ConvertToAddress(long value) =>
        ↪ _int64ToAddressConverter.Convert(value);

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

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

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

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

        #region Disposable

        protected override bool AllowMultipleDisposeCalls
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                ResetPointers();
                _memory.DisposeIfPossible();
            }
        }

        #endregion
    }
}
```

1.48 ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnusedLinksListMethods.cs

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

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

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
    ↪  ILinksListMethods<TLink>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
        ↪  UncheckedConverter<TLink, long>.Default;

        private readonly byte* _links;
        private readonly byte* _header;

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

        [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 *
        ↪  _addressToInt64Converter.Convert(link));

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

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

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

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

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

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

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

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

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

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

1.49 ./csharp/Platform.Data.Doublets/Memory/United/RawLink.cs

```
using Platform.Unsafe;
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.Memory.United
```

```
{
    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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is RawLink<TLink> link ? Equals(link) :
        ↪  false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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);

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator ==(RawLink<TLink> left, RawLink<TLink> right) =>
        ↪  left.Equals(right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator !=(RawLink<TLink> left, RawLink<TLink> right) => !(left ==
        ↪  right);
    }
}
```

1.50 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs

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

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

namespace Platform.Data.Doublets.Memory.United.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;
```

```
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override bool GreaterThan(ulong first, ulong second) => first > second;
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
38
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) => (value & 4294967264UL) >> 5;
84
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
            ↪  storedValue & 31UL | (size & 134217727UL) << 5;
87
88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
89          protected override bool GetLeftIsChildValue(ulong value) => (value & 16UL) >> 4 == 1UL;
90
91          [MethodImpl(MethodImplOptions.AggressiveInlining)]
92          protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
            ↪  storedValue = storedValue & 4294967279UL | (As<bool, byte>(ref value) & 1UL) << 4;
93
94          [MethodImpl(MethodImplOptions.AggressiveInlining)]
95          protected override bool GetRightIsChildValue(ulong value) => (value & 8UL) >> 3 == 1UL;
96
97          [MethodImpl(MethodImplOptions.AggressiveInlining)]
98          protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>
            ↪  storedValue = 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];
    }
}
```

1.51 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs

```
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.Memory.United.Generic;

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

namespace Platform.Data.Doublets.Memory.United.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];
    }
}
```

1.52 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

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

```

        //[MethodImpl(MethodImplOptions.AggressiveInlining)]
        //protected override bool GetLeftIsChild(ulong node) => IsChild(node, GetLeft(node));

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

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

        //[MethodImpl(MethodImplOptions.AggressiveInlining)]
        //protected override bool GetRightIsChild(ulong node) => IsChild(node, GetRight(node));

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

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

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

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

        [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;
        }
    }
}
```

1.53 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Memory.United.Specific
{
    public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
    ↪  UInt64LinksSizeBalancedTreeMethodsBase
    {
        public UInt64LinksSourcesSizeBalancedTreeMethods(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)]
```

```
18          protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
            ↪  left;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
            ↪  right;
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override ulong GetSize(ulong node) => Links[node].SizeAsSource;
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsSource =
            ↪  size;
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected override ulong GetTreeRoot() => Header->RootAsSource;
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
            ↪  ulong secondSource, ulong secondTarget)
43              => firstSource < secondSource || firstSource == secondSource && firstTarget <
                ↪  secondTarget;
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
            ↪  ulong secondSource, ulong secondTarget)
47              => firstSource > secondSource || firstSource == secondSource && firstTarget >
                ↪  secondTarget;
48
49          [MethodImpl(MethodImplOptions.AggressiveInlining)]
50          protected override void ClearNode(ulong node)
51          {
52              ref var link = ref Links[node];
53              link.LeftAsSource = 0UL;
54              link.RightAsSource = 0UL;
55              link.SizeAsSource = 0UL;
56          }
57      }
58  }
```

1.54 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Data.Doublets.Memory.United.Specific
6   {
7       public unsafe class UInt64LinksTargetsAvlBalancedTreeMethods :
        ↪  UInt64LinksAvlBalancedTreeMethodsBase
8       {
9           public UInt64LinksTargetsAvlBalancedTreeMethods(LinksConstants<ulong> constants,
            ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
            ↪  { }
10
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected override ref ulong GetLeftReference(ulong node) => ref
            ↪  Links[node].LeftAsTarget;
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          protected override ref ulong GetRightReference(ulong node) => ref
            ↪  Links[node].RightAsTarget;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
24          protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
            ↪ left;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
            ↪ right;
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsTarget);
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->RootAsTarget;
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 = OUL;
72              link.RightAsTarget = OUL;
73              link.SizeAsTarget = OUL;
74          }
75      }
76  }
```

1.55 ./csharp/Platform.Data.Doublets/Memory/United/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.Memory.United.Specific
6   {
7       public unsafe class UInt64LinksTargetsSizeBalancedTreeMethods :
        ↪ UInt64LinksSizeBalancedTreeMethodsBase
8       {
```

```
        public UInt64LinksTargetsSizeBalancedTreeMethods(LinksConstants<ulong> constants,
        ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
        ↪  { }

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

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

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

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

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

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

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

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

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

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

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

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

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

1.56 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnitedMemoryLinks.cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Memory;
using Platform.Singletons;
using Platform.Data.Doublets.Memory.United.Generic;

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

namespace Platform.Data.Doublets.Memory.United.Specific
{
    /// <summary>
    /// <para>Represents a low-level implementation of direct access to resizable memory, for
    ↪  organizing the storage of links with addresses represented as <see cref="ulong"
    ↪  />.</para>
```

```
13      /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
        ↪  размером, для организации хранения связей с адресами представленными в виде <see
        ↪  cref="ulong"/>.</para>
14      /// </summary>
15      public unsafe class UInt64UnitedMemoryLinks : UnitedMemoryLinksBase<ulong>
16      {
17          private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
18          private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
19          private LinksHeader<ulong>* _header;
20          private RawLink<ulong>* _links;
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          public UInt64UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
24
25          /// <summary>
26          /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
            ↪  минимальным шагом расширения базы данных.
27          /// </summary>
28          /// <param name="address">Полный пусть к файлу базы данных.</param>
29          /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            ↪  байтах.</param>
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public UInt64UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
            ↪  FileMappedResizableDirectMemory(address, memoryReservationStep),
            ↪  memoryReservationStep) { }
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public UInt64UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
            ↪  DefaultLinksSizeStep) { }
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep) : this(memory, memoryReservationStep,
            ↪  Default<LinksConstants<ulong>>.Instance, true) { }
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep, LinksConstants<ulong> constants, bool useAvlBasedIndex) :
            ↪  base(memory, memoryReservationStep, constants)
41          {
42              if (useAvlBasedIndex)
43              {
44                  _createSourceTreeMethods = () => new
                    ↪  UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
45                  _createTargetTreeMethods = () => new
                    ↪  UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
46              }
47              else
48              {
49                  _createSourceTreeMethods = () => new
                    ↪  UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
50                  _createTargetTreeMethods = () => new
                    ↪  UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
51              }
52              Init(memory, memoryReservationStep);
53          }
54
55          [MethodImpl(MethodImplOptions.AggressiveInlining)]
56          protected override void SetPointers(IResizableDirectMemory memory)
57          {
58              _header = (LinksHeader<ulong>*)memory.Pointer;
59              _links = (RawLink<ulong>*)memory.Pointer;
60              SourcesTreeMethods = _createSourceTreeMethods();
61              TargetsTreeMethods = _createTargetTreeMethods();
62              UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
63          }
64
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          protected override void ResetPointers()
67          {
68              base.ResetPointers();
69              _links = null;
70              _header = null;
71          }
72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
75
```

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override long ConvertToInt64(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;
    }
}
```

1.57 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnusedLinksListMethods.cs

```
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.Memory.United.Generic;

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

namespace Platform.Data.Doublets.Memory.United.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;
    }
}
```

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

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

```

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
        ↪  powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
        ↪  powerOf2ToUnaryNumberConverter;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(TLink number)
        {
            var links = _links;
            var nullConstant = links.Constants.Null;
            var target = nullConstant;
            for (var i = 0; !_equalityComparer.Equals(number, _zero) && 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;
        }
    }
}
```

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

```

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

#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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkToItsFrequencyNumberConveter(
            ILinks<TLink> links,
            IProperty<TLink, TLink> frequencyPropertyOperator,
            IConverter<TLink> unaryNumberToAddressConverter)
            : base(links)
        {
            _frequencyPropertyOperator = frequencyPropertyOperator;
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(Doublet<TLink> doublet)
        {
            var links = _links;
            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);
        }
    }
}
```

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

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

#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;

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

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

1.61 ./csharp/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 static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
        ↪ UncheckedConverter<TLink, ulong>.Default;
        private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
        ↪ UncheckedConverter<ulong, TLink>.Default;
        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

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

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
            : base(links)
        {
            _unaryOne = unaryOne;
            _unaryToUInt64 = CreateUnaryToUInt64Dictionary(links, unaryOne);
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Dictionary<TLink, TLink> CreateUnaryToUInt64Dictionary(ILinks<TLink>
        ↪  links, TLink unaryOne)
        {
            var 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);
            }
            return unaryToUInt64;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink Double(TLink number) =>
        ↪  _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Reflection;
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 UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<TLink>
```

```
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
        ↪  TLink> powerOf2ToUnaryNumberConverter) : base(links) => _unaryNumberPowerOf2Indicies
        ↪  = CreateUnaryNumberPowerOf2IndiciesDictionary(powerOf2ToUnaryNumberConverter);

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Dictionary<TLink, int>
        ↪  CreateUnaryNumberPowerOf2IndiciesDictionary(IConverter<int, TLink>
        ↪  powerOf2ToUnaryNumberConverter)
        {
            var unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
            for (int i = 0; i < NumericType<TLink>.BitsSize; i++)
            {
                unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
            }
            return unaryNumberPowerOf2Indicies;
        }

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink GetValue(TLink @object, TLink property)
```

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void SetValue(TLink @object, TLink property, TLink value)
        {
            var links = _links;
            var objectProperty = links.GetOrCreate(@object, property);
            links.DeleteMany(links.AllIndices(links.Constants.Any, objectProperty));
            links.GetOrCreate(objectProperty, value);
        }
    }
}
```

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
        ↪  propertyValueMarker) : base(links)
        {
            _propertyMarker = propertyMarker;
            _propertyValueMarker = propertyValueMarker;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Get(TLink link)
        {
            var property = _links.SearchOrDefault(link, _propertyMarker);
            return GetValue(GetContainer(property));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink GetContainer(TLink property)
        {
            var valueContainer = default(TLink);
            if (_equalityComparer.Equals(property, default))
            {
                return valueContainer;
            }
            var links = _links;
            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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
        ↪  ? default : _links.GetTarget(container);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Set(TLink link, TLink value)
        {
            var links = _links;
            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);
            }
        }
    }
}
```

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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]);
        }

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

1.66 ./csharp/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 static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        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;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
            {
                Element = element;
                DoubletData = doubletData;
            }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink> baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
            : this(links, baseConverter, doubletFrequenciesCache, _one, true) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink> baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool doInitialFrequenciesIncrement)
            : this(links, baseConverter, doubletFrequenciesCache, _one, doInitialFrequenciesIncrement) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink> baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink minFrequencyToCompress, bool doInitialFrequenciesIncrement)
```

```
60              : base(links)
61          {
62              _baseConverter = baseConverter;
63              _doubletFrequenciesCache = doubletFrequenciesCache;
64              if (_comparer.Compare(minFrequencyToCompress, _one) < 0)
65              {
66                  minFrequencyToCompress = _one;
67              }
68              _minFrequencyToCompress = minFrequencyToCompress;
69              _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
70              ResetMaxDoublet();
71          }
72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          public override TLink Convert(IList<TLink> source) =>
            ↪  _baseConverter.Convert(Compress(source));
75
76          /// <remarks>
77          /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding .
78          /// Faster version (doublets' frequencies dictionary is not recreated).
79          /// </remarks>
80          [MethodImpl(MethodImplOptions.AggressiveInlining)]
81          private IList<TLink> Compress(IList<TLink> sequence)
82          {
83              if (sequence.IsNullOrEmpty())
84              {
85                  return null;
86              }
87              if (sequence.Count == 1)
88              {
89                  return sequence;
90              }
91              if (sequence.Count == 2)
92              {
93                  return new[] { _links.GetOrCreate(sequence[0], sequence[1]) };
94              }
95              // TODO: arraypool with min size (to improve cache locality) or stackallow with Sigil
96              var copy = new HalfDoublet[sequence.Count];
97              Doublet<TLink> doublet = default;
98              for (var i = 1; i < sequence.Count; i++)
99              {
100                 doublet.Source = sequence[i - 1];
101                 doublet.Target = sequence[i];
102                 LinkFrequency<TLink> data;
103                 if (_doInitialFrequenciesIncrement)
104                 {
105                     data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
106                 }
107                 else
108                 {
109                     data = _doubletFrequenciesCache.GetFrequency(ref doublet);
110                     if (data == null)
111                     {
112                         throw new NotSupportedException("If you ask not to increment
                            ↪  frequencies, it is expected that all frequencies for the sequence
                            ↪  are prepared.");
113                     }
114                 }
115                 copy[i - 1].Element = sequence[i - 1];
116                 copy[i - 1].DoubletData = data;
117                 UpdateMaxDoublet(ref doublet, data);
118             }
119             copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
120             copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
121             if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
122             {
123                 var newLength = ReplaceDoublets(copy);
124                 sequence = new TLink[newLength];
125                 for (int i = 0; i < newLength; i++)
126                 {
127                     sequence[i] = copy[i].Element;
128                 }
129             }
130             return sequence;
131         }
132
133         /// <remarks>
134         /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding
135         /// </remarks>
```

```
136         [MethodImpl(MethodImplOptions.AggressiveInlining)]
137         private int ReplaceDoublets(HalfDoublet[] copy)
138         {
139             var oldLength = copy.Length;
140             var newLength = copy.Length;
141             while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
142             {
143                 var maxDoubletSource = _maxDoublet.Source;
144                 var maxDoubletTarget = _maxDoublet.Target;
145                 if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
146                 {
147                     _maxDoubletData.Link = _links.GetOrCreate(maxDoubletSource,
                        ↪ maxDoubletTarget);
148                 }
149                 var maxDoubletReplacementLink = _maxDoubletData.Link;
150                 oldLength--;
151                 var oldLengthMinusTwo = oldLength - 1;
152                 // Substitute all usages
153                 int w = 0, r = 0; // (r == read, w == write)
154                 for (; r < oldLength; r++)
155                 {
156                     if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
                        ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
157                     {
158                         if (r > 0)
159                         {
160                             var previous = copy[w - 1].Element;
161                             copy[w - 1].DoubletData.DecrementFrequency();
162                             copy[w - 1].DoubletData =
                                ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
                                ↪ maxDoubletReplacementLink);
163                         }
164                         if (r < oldLengthMinusTwo)
165                         {
166                             var next = copy[r + 2].Element;
167                             copy[r + 1].DoubletData.DecrementFrequency();
168                             copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(ma
                                ↪ xDoubletReplacementLink,
                                ↪ next);
169                         }
170                         copy[w++].Element = maxDoubletReplacementLink;
171                         r++;
172                         newLength--;
173                     }
174                     else
175                     {
176                         copy[w++] = copy[r];
177                     }
178                 }
179                 if (w < newLength)
180                 {
181                     copy[w] = copy[r];
182                 }
183                 oldLength = newLength;
184                 ResetMaxDoublet();
185                 UpdateMaxDoublet(copy, newLength);
186             }
187             return newLength;
188         }
189 
190         [MethodImpl(MethodImplOptions.AggressiveInlining)]
191         private void ResetMaxDoublet()
192         {
193             _maxDoublet = new Doublet<TLink>();
194             _maxDoubletData = new LinkFrequency<TLink>();
195         }
196 
197         [MethodImpl(MethodImplOptions.AggressiveInlining)]
198         private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
199         {
200             Doublet<TLink> doublet = default;
201             for (var i = 1; i < length; i++)
202             {
203                 doublet.Source = copy[i - 1].Element;
204                 doublet.Target = copy[i].Element;
205                 UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
206             }
207         }
208 
```

```
209             [MethodImpl(MethodImplOptions.AggressiveInlining)]
210             private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
211             {
212                 var frequency = data.Frequency;
213                 var maxFrequency = _maxDoubletData.Frequency;
214                 //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
                    ↪  (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
                    ↪  compression string data (and gives collisions quickly) */ _maxDoublet.Source +
                    ↪  _maxDoublet.Target)))
215                 if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
216                    (_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) */
217                 {
218                     _maxDoublet = doublet;
219                     _maxDoubletData = data;
220                 }
221             }
222         }
223     }
```

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

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Converters;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Sequences.Converters
8   {
9       public abstract class LinksListToSequenceConverterBase<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<IList<TLink>, TLink>
10      {
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected LinksListToSequenceConverterBase(ILinks<TLink> links) : base(links) { }
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          public abstract TLink Convert(IList<TLink> source);
16      }
17  }
```

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
        ↪ _comparer.Compare(next, current) < 0 ? next : current;
```

```
107
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
            ↪  => _comparer.Compare(previous, current) < 0 ? previous : current;
110     }
111 }
```

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

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Converters;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences.Converters
8  {
9      public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
       ↪  IConverter<IList<TLink>>
10     {
11         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
12
13         private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
           ↪  IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
           ↪  => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public IList<TLink> Convert(IList<TLink> sequence)
20         {
21             var levels = new TLink[sequence.Count];
22             levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
23             for (var i = 1; i < sequence.Count - 1; i++)
24             {
25                 var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
26                 var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
27                 levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
28             }
29             levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
               ↪  sequence[sequence.Count - 1]);
30             return levels;
31         }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public TLink GetFrequencyNumber(TLink source, TLink target) =>
           ↪  _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
35     }
36 }
```

1.70 ./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs

```
1  using System.Runtime.CompilerServices;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.CriterionMatchers
7  {
8      public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
       ↪  ICriterionMatcher<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public bool IsMatched(TLink argument) => _links.IsPartialPoint(argument);
15     }
16 }
```

1.71 ./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
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.CriterionMatchers
8  {
9      public class MarkedSequenceCriterionMatcher<TLink> : ICriterionMatcher<TLink>
```

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
        {
            _links = links;
            _sequenceMarkerLink = sequenceMarkerLink;
        }

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
        ↪ ISequenceHeightProvider<TLink> heightProvider)
            : base(links)
        {
            _stack = stack;
            _heightProvider = heightProvider;
        }

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

```
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences
{
    public class DuplicateSegmentsCounter<TLink> : ICounter<int>
    {
        private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
        ↪  _duplicateFragmentsProvider;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
        ↪  IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
        ↪  duplicateFragmentsProvider;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences
{
    public class DuplicateSegmentsProvider<TLink> :
    ↪  DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
    ↪  IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
        ↪  UncheckedConverter<TLink, long>.Default;
        private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
        ↪  UncheckedConverter<TLink, ulong>.Default;
        private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
        ↪  UncheckedConverter<ulong, TLink>.Default;

        private readonly ILinks<TLink> _links;
        private readonly ILinks<TLink> _sequences;
        private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
        private BitString _visited;

        private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
        ↪  IList<TLink>>>
        {
            private readonly IListEqualityComparer<TLink> _listComparer;

            public ItemEquilityComparer() => _listComparer =
            ↪  Default<IListEqualityComparer<TLink>>.Instance;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            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);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
            ↪  (_listComparer.GetHashCode(pair.Key),
            ↪  _listComparer.GetHashCode(pair.Value)).GetHashCode();
        }
```

```
41
42          private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
43          {
44              private readonly IListComparer<TLink> _listComparer;
45
46              [MethodImpl(MethodImplOptions.AggressiveInlining)]
47              public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
48
49              [MethodImpl(MethodImplOptions.AggressiveInlining)]
50              public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
                ↪  KeyValuePair<IList<TLink>, IList<TLink>> right)
51              {
52                  var intermediateResult = _listComparer.Compare(left.Key, right.Key);
53                  if (intermediateResult == 0)
54                  {
55                      intermediateResult = _listComparer.Compare(left.Value, right.Value);
56                  }
57                  return intermediateResult;
58              }
59          }
60
61          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
63              : base(minimumStringSegmentLength: 2)
64          {
65              _links = links;
66              _sequences = sequences;
67          }
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
71          {
72              _groups = new HashSet<KeyValuePair<IList<TLink>,
                ↪  IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
73              var links = _links;
74              var count = links.Count();
75              _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
76              links.Each(link =>
77              {
78                  var linkIndex = links.GetIndex(link);
79                  var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
80                  var constants = links.Constants;
81                  if (!_visited.Get(linkBitIndex))
82                  {
83                      var sequenceElements = new List<TLink>();
84                      var filler = new ListFiller<TLink, TLink>(sequenceElements, constants.Break);
85                      _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
                        ↪  LinkAddress<TLink>(linkIndex));
86                      if (sequenceElements.Count > 2)
87                      {
88                          WalkAll(sequenceElements);
89                      }
90                  }
91                  return constants.Continue;
92              });
93              var resultList = _groups.ToList();
94              var comparer = Default<ItemComparer>.Instance;
95              resultList.Sort(comparer);
96  #if DEBUG
97              foreach (var item in resultList)
98              {
99                  PrintDuplicates(item);
100             }
101 #endif
102             return resultList;
103         }
104
105         [MethodImpl(MethodImplOptions.AggressiveInlining)]
106         protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
            ↪  length) => new Segment<TLink>(elements, offset, length);
107
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         protected override void OnDublicateFound(Segment<TLink> segment)
110         {
111             var duplicates = CollectDuplicatesForSegment(segment);
112             if (duplicates.Count > 1)
113             {
114                 _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
                    ↪  duplicates));
```

```
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
        {
            var duplicates = new List<TLink>();
            var readAsElement = new HashSet<TLink>();
            var restrictions = segment.ShiftRight();
            var constants = _links.Constants;
            restrictions[0] = constants.Any;
            _sequences.Each(sequence =>
            {
                var sequenceIndex = sequence[constants.IndexPart];
                duplicates.Add(sequenceIndex);
                readAsElement.Add(sequenceIndex);
                return constants.Continue;
            }, restrictions);
            if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
            {
                return new List<TLink>();
            }
            foreach (var duplicate in duplicates)
            {
                var duplicateBitIndex = _addressToInt64Converter.Convert(duplicate);
                _visited.Set(duplicateBitIndex);
            }
            if (_sequences is Sequences sequencesExperiments)
            {
                var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
                ↪  ashSet<ulong>)(object)readAsElement,
                ↪  (IList<ulong>)segment);
                foreach (var partiallyMatchedSequence in partiallyMatched)
                {
                    var sequenceIndex =
                    ↪  _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
                    duplicates.Add(sequenceIndex);
                }
            }
            duplicates.Sort();
            return duplicates;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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++)
            {
                var sequenceIndex = _addressToUInt64Converter.Convert(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();
        }
    }
}
```

1.75 ./csharp/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 static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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);
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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>(_one, link);
                if (!_equalityComparer.Equals(link, default))
                {
                    data.Frequency = Arithmetic.Add(data.Frequency,
                    ↪  _frequencyCounter.Count(link));
                }
                _doubletsCache.Add(doublet, data);
            }
            return data;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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), _one) > 0))
                     || ((_comparer.Compare(count, frequency) > 0) &&
                     ↪  (_comparer.Compare(Arithmetic.Subtract(count, frequency), _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 InvalidOperationException("Frequencies validation
                ↪  failed.");
                //    }
                //}
            }
        }
    }
}
```

1.76 ./csharp/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; }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency(TLink frequency, TLink link)
        {
            Frequency = frequency;
            Link = link;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency() { }
```

```

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override string ToString() => $"F: {Frequency}, L: {Link}";
    }
}
```

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

```
using System.Runtime.CompilerServices;
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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public
        ↪ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
        ↪ cache) => _cache = cache;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
    }
}
```

1.78 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOf

```
using System.Runtime.CompilerServices;
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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
        ↪ ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
            : base(links, sequenceLink, symbol)
            => _markedSequenceMatcher = markedSequenceMatcher;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Count()
        {
            if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
            {
                return default;
            }
            return base.Count();
        }
    }
}
```

1.79 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounte

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

```
13          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
14          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
15
16          protected readonly ILinks<TLink> _links;
17          protected readonly TLink _sequenceLink;
18          protected readonly TLink _symbol;
19          protected TLink _total;
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
            ↪ TLink symbol)
23          {
24              _links = links;
25              _sequenceLink = sequenceLink;
26              _symbol = symbol;
27              _total = default;
28          }
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public virtual TLink Count()
32          {
33              if (_comparer.Compare(_total, default) > 0)
34              {
35                  return _total;
36              }
37              StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
                ↪ IsElement, VisitElement);
38              return _total;
39          }
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
            ↪ _links.IsPartialPoint(x); // TODO: Use SequenceElementCreteriaMatcher instead of
            ↪ IsPartialPoint
43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          private bool VisitElement(TLink element)
46          {
47              if (_equalityComparer.Equals(element, _symbol))
48              {
49                  _total = Arithmetic.Increment(_total);
50              }
51              return true;
52          }
53      }
54  }
```

1.80 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyC

```
1   using System.Runtime.CompilerServices;
2   using Platform.Interfaces;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7   {
8       public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
9       {
10          private readonly ILinks<TLink> _links;
11          private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
            ↪ ICriterionMatcher<TLink> markedSequenceMatcher)
15          {
16              _links = links;
17              _markedSequenceMatcher = markedSequenceMatcher;
18          }
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          public TLink Count(TLink argument) => new
            ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
            ↪ _markedSequenceMatcher, argument).Count();
22      }
23  }
```

1.81 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyC

```
1   using System.Runtime.CompilerServices;
2   using Platform.Interfaces;
3   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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
        ↪ ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
            : base(links, symbol)
            => _markedSequenceMatcher = markedSequenceMatcher;

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

1.82 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.

```

using System.Runtime.CompilerServices;
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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Count(TLink symbol) => new
        ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
    }
}
```

1.83 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffC

```

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.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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
        {
            _links = links;
            _symbol = symbol;
            _visits = new HashSet<TLink>();
            _total = default;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Count()
```

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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);
            }
        }

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CachedSequenceHeightProvider(
            ISequenceHeightProvider<TLink> baseHeightProvider,
            IConverter<TLink> addressToUnaryNumberConverter,
            IConverter<TLink> unaryNumberToAddressConverter,
            TLink heightPropertyMarker,
            IProperties<TLink, TLink, TLink> propertyOperator)
        {
            _heightPropertyMarker = heightPropertyMarker;
            _baseHeightProvider = baseHeightProvider;
            _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
            _propertyOperator = propertyOperator;
        }

```

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

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

```
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.HeightProviders
{
    public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
    ↪ ISequenceHeightProvider<TLink>
    {
        private readonly ICriterionMatcher<TLink> _elementMatcher;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
        ↪ elementMatcher) : base(links) => _elementMatcher = elementMatcher;

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

1.86 ./csharp/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>
    {
    }
}
```

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

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

```

```
13          private readonly LinkFrequenciesCache<TLink> _cache;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
            ↪  _cache = cache;
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          public bool Add(IList<TLink> sequence)
20          {
21              var indexed = true;
22              var i = sequence.Count;
23              while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
                ↪  { }
24              for (; i >= 1; i--)
25              {
26                  _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
27              }
28              return indexed;
29          }
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          private bool IsIndexedWithIncrement(TLink source, TLink target)
33          {
34              var frequency = _cache.GetFrequency(source, target);
35              if (frequency == null)
36              {
37                  return false;
38              }
39              var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
40              if (indexed)
41              {
42                  _cache.IncrementFrequency(source, target);
43              }
44              return indexed;
45          }
46
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          public bool MightContain(IList<TLink> sequence)
49          {
50              var indexed = true;
51              var i = sequence.Count;
52              while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
53              return indexed;
54          }
55
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          private bool IsIndexed(TLink source, TLink target)
58          {
59              var frequency = _cache.GetFrequency(source, target);
60              if (frequency == null)
61              {
62                  return false;
63              }
64              return !_equalityComparer.Equals(frequency.Frequency, default);
65          }
66      }
67  }
```

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

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Interfaces;
4   using Platform.Incrementers;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Sequences.Indexes
9   {
10      public class FrequencyIncrementingSequenceIndex<TLink> : SequenceIndex<TLink>,
        ↪  ISequenceIndex<TLink>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
13
14          private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
15          private readonly IIncrementer<TLink> _frequencyIncrementer;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IProperty<TLink, TLink>
            ↪  frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
```

```
            : base(links)
        {
            _frequencyPropertyOperator = frequencyPropertyOperator;
            _frequencyIncrementer = frequencyIncrementer;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

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

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

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

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

namespace Platform.Data.Doublets.Sequences.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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        bool Add(IList<TLink> sequence);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        bool MightContain(IList<TLink> sequence);
    }
}
```

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
13          public SequenceIndex(ILinks<TLink> links) : base(links) { }
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public virtual bool Add(IList<TLink> sequence)
17          {
18              var indexed = true;
19              var i = sequence.Count;
20              while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
                ↪  default))) { }
21              for (; i >= 1; i--)
22              {
23                  _links.GetOrCreate(sequence[i - 1], sequence[i]);
24              }
25              return indexed;
26          }
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          public virtual bool MightContain(IList<TLink> sequence)
30          {
31              var indexed = true;
32              var i = sequence.Count;
33              while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
                ↪  default))) { }
34              return indexed;
35          }
36      }
37  }
```

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

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Sequences.Indexes
7   {
8       public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
9       {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
11
12          private readonly ISynchronizedLinks<TLink> _links;
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          public bool Add(IList<TLink> sequence)
19          {
20              var indexed = true;
21              var i = sequence.Count;
22              var links = _links.Unsync;
23              _links.SyncRoot.ExecuteReadOperation(() =>
24              {
25                  while (--i >= 1 && (indexed =
                    ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                    ↪  sequence[i]), default))) { }
26              });
27              if (!indexed)
28              {
29                  _links.SyncRoot.ExecuteWriteOperation(() =>
30                  {
31                      for (; i >= 1; i--)
32                      {
33                          links.GetOrCreate(sequence[i - 1], sequence[i]);
34                      }
35                  });
36              }
37              return indexed;
38          }
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          public bool MightContain(IList<TLink> sequence)
42          {
43              var links = _links.Unsync;
44              return _links.SyncRoot.ExecuteReadOperation(() =>
45              {
```

```
                var indexed = true;
                var i = sequence.Count;
                while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                ↪  sequence[i]), default))) { }
                return indexed;
            });
        }
    }
}
```

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool MightContain(IList<TLink> sequence) => true;
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
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;
using LinkIndex = System.UInt64;
using Stack = System.Collections.Generic.Stack<ulong>;

#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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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);
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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);
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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);
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> EachPart(params ulong[] sequence)
        {
```

```
202             var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
203             EachPartCore(link =>
204             {
205                 var linkIndex = link[Constants.IndexPart];
206                 if (!visitedLinks.Contains(linkIndex))
207                 {
208                     visitedLinks.Add(linkIndex); // изучить почему случаются повторы
209                 }
210                 return Constants.Continue;
211             }, sequence);
212             return visitedLinks;
213         }
214 
215         [MethodImpl(MethodImplOptions.AggressiveInlining)]
216         public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
217         {
218             var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
219             EachPartCore(link =>
220             {
221                 var linkIndex = link[Constants.IndexPart];
222                 if (!visitedLinks.Contains(linkIndex))
223                 {
224                     visitedLinks.Add(linkIndex); // изучить почему случаются повторы
225                     return handler(new LinkAddress<LinkIndex>(linkIndex));
226                 }
227                 return Constants.Continue;
228             }, sequence);
229         }
230 
231         [MethodImpl(MethodImplOptions.AggressiveInlining)]
232         private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
    ↪ sequence)
233         {
234             if (sequence.IsNullOrEmpty())
235             {
236                 return;
237             }
238             Links.EnsureLinkIsAnyOrExists(sequence);
239             if (sequence.Length == 1)
240             {
241                 var link = sequence[0];
242                 if (link > 0)
243                 {
244                     handler(new LinkAddress<LinkIndex>(link));
245                 }
246                 else
247                 {
248                     Links.Each(Constants.Any, Constants.Any, handler);
249                 }
250             }
251             else if (sequence.Length == 2)
252             {
253                 //_links.Each(sequence[0], sequence[1], handler);
254                 //  o_|      x_o ...
255                 // x_|        |___|
256                 Links.Each(sequence[1], Constants.Any, doublet =>
257                 {
258                     var match = Links.SearchOrDefault(sequence[0], doublet);
259                     if (match != Constants.Null)
260                     {
261                         handler(new LinkAddress<LinkIndex>(match));
262                     }
263                     return true;
264                 });
265                 // |_x      ... x_o
266                 //  |_o      |___|
267                 Links.Each(Constants.Any, sequence[0], doublet =>
268                 {
269                     var match = Links.SearchOrDefault(doublet, sequence[1]);
270                     if (match != 0)
271                     {
272                         handler(new LinkAddress<LinkIndex>(match));
273                     }
274                     return true;
275                 });
276                 //           ._x o_.
277                 //            |___|
278                 PartialStepRight(x => handler(x), sequence[0], sequence[1]);
```

```
279             }
280             else
281             {
282                 throw new NotImplementedException();
283             }
284         }
285 
286         [MethodImpl(MethodImplOptions.AggressiveInlining)]
287         private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
288         {
289             Links.Unsync.Each(Constants.Any, left, doublet =>
290             {
291                 StepRight(handler, doublet, right);
292                 if (left != doublet)
293                 {
294                     PartialStepRight(handler, doublet, right);
295                 }
296                 return true;
297             });
298         }
299 
300         [MethodImpl(MethodImplOptions.AggressiveInlining)]
301         private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
302         {
303             Links.Unsync.Each(left, Constants.Any, rightStep =>
304             {
305                 TryStepRightUp(handler, right, rightStep);
306                 return true;
307             });
308         }
309 
310         [MethodImpl(MethodImplOptions.AggressiveInlining)]
311         private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
            ↪ stepFrom)
312         {
313             var upStep = stepFrom;
314             var firstSource = Links.Unsync.GetTarget(upStep);
315             while (firstSource != right && firstSource != upStep)
316             {
317                 upStep = firstSource;
318                 firstSource = Links.Unsync.GetSource(upStep);
319             }
320             if (firstSource == right)
321             {
322                 handler(new LinkAddress<LinkIndex>(stepFrom));
323             }
324         }
325 
326         // TODO: Test
327         [MethodImpl(MethodImplOptions.AggressiveInlining)]
328         private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
329         {
330             Links.Unsync.Each(right, Constants.Any, doublet =>
331             {
332                 StepLeft(handler, left, doublet);
333                 if (right != doublet)
334                 {
335                     PartialStepLeft(handler, left, doublet);
336                 }
337                 return true;
338             });
339         }
340 
341         [MethodImpl(MethodImplOptions.AggressiveInlining)]
342         private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
343         {
344             Links.Unsync.Each(Constants.Any, right, leftStep =>
345             {
346                 TryStepLeftUp(handler, left, leftStep);
347                 return true;
348             });
349         }
350 
351         [MethodImpl(MethodImplOptions.AggressiveInlining)]
352         private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
353         {
354             var upStep = stepFrom;
355             var firstTarget = Links.Unsync.GetSource(upStep);
356             while (firstTarget != left && firstTarget != upStep)
```

```
            {
                upStep = firstTarget;
                firstTarget = Links.Unsync.GetTarget(upStep);
            }
            if (firstTarget == left)
            {
                handler(new LinkAddress<LinkIndex>(stepFrom));
            }
        }

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

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

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

```
                            }
                            filterPosition++;

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

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

    public const int MaxSequenceFormatSize = 200;

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

```
509
510         [MethodImpl(MethodImplOptions.AggressiveInlining)]
511         public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
            ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
            ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
            ↪ elementToString, insertComma, knownElements));
512
513         [MethodImpl(MethodImplOptions.AggressiveInlining)]
514         private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
            ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
            ↪ LinkIndex[] knownElements)
515         {
516             var linksInSequence = new HashSet<ulong>(knownElements);
517             //var entered = new HashSet<ulong>();
518             var sb = new StringBuilder();
519             sb.Append('{');
520             if (links.Exists(sequenceLink))
521             {
522                 StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
523                     x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
                        ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
524                     {
525                         if (insertComma && sb.Length > 1)
526                         {
527                             sb.Append(',');
528                         }
529                         //if (entered.Contains(element))
530                         //{
531                         //    sb.Append('{');
532                         //    elementToString(sb, element);
533                         //    sb.Append('}');
534                         //}
535                         //else
536                         elementToString(sb, element);
537                         if (sb.Length < MaxSequenceFormatSize)
538                         {
539                             return true;
540                         }
541                         sb.Append(insertComma ? ", ..." : "...");
542                         return false;
543                     });
544             }
545             sb.Append('}');
546             return sb.ToString();
547         }
548
549         [MethodImpl(MethodImplOptions.AggressiveInlining)]
550         public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
            ↪ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
            ↪ knownElements);
551
552         [MethodImpl(MethodImplOptions.AggressiveInlining)]
553         public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
            ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
            ↪ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
            ↪ sequenceLink, elementToString, insertComma, knownElements));
554
555         [MethodImpl(MethodImplOptions.AggressiveInlining)]
556         private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
            ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
            ↪ LinkIndex[] knownElements)
557         {
558             var linksInSequence = new HashSet<ulong>(knownElements);
559             var entered = new HashSet<ulong>();
560             var sb = new StringBuilder();
561             sb.Append('{');
562             if (links.Exists(sequenceLink))
563             {
564                 StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
565                     x => linksInSequence.Contains(x) || links.IsFullPoint(x),
                        ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
566                     {
567                         if (insertComma && sb.Length > 1)
568                         {
569                             sb.Append(',');
570                         }
571                         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();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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>();
            });
        }

```

```
649         [MethodImpl(MethodImplOptions.AggressiveInlining)]
650         public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
651         {
652             return _sync.ExecuteReadOperation(() =>
653             {
654                 if (sequence.Length > 0)
655                 {
656                     Links.EnsureLinkExists(sequence);
657                     var results = new HashSet<ulong>();
658                     for (var i = 0; i < sequence.Length; i++)
659                     {
660                         AllUsagesCore(sequence[i], results);
661                     }
662                     var filteredResults = new HashSet<ulong>();
663                     var matcher = new Matcher(this, sequence, filteredResults, null);
664                     matcher.AddAllPartialMatchedToResults(results);
665                     return filteredResults;
666                 }
667                 return new HashSet<ulong>();
668             });
669         }
670 
671         [MethodImpl(MethodImplOptions.AggressiveInlining)]
672         public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
            ↪ params ulong[] sequence)
673         {
674             return _sync.ExecuteReadOperation(() =>
675             {
676                 if (sequence.Length > 0)
677                 {
678                     Links.EnsureLinkExists(sequence);
679 
680                     var results = new HashSet<ulong>();
681                     var filteredResults = new HashSet<ulong>();
682                     var matcher = new Matcher(this, sequence, filteredResults, handler);
683                     for (var i = 0; i < sequence.Length; i++)
684                     {
685                         if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
686                         {
687                             return false;
688                         }
689                     }
690                     return true;
691                 }
692                 return true;
693             });
694         }
695 
696         //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
697         //{
698         //    return Sync.ExecuteReadOperation(() =>
699         //    {
700         //        if (sequence.Length > 0)
701         //        {
702         //            _links.EnsureEachLinkIsAnyOrExists(sequence);
703 
704         //            var firstResults = new HashSet<ulong>();
705         //            var lastResults = new HashSet<ulong>();
706 
707         //            var first = sequence.First(x => x != LinksConstants.Any);
708         //            var last = sequence.Last(x => x != LinksConstants.Any);
709 
710         //            AllUsagesCore(first, firstResults);
711         //            AllUsagesCore(last, lastResults);
712 
713         //            firstResults.IntersectWith(lastResults);
714 
715         //            //for (var i = 0; i < sequence.Length; i++)
716         //            //    AllUsagesCore(sequence[i], results);
717 
718         //            var filteredResults = new HashSet<ulong>();
719         //            var matcher = new Matcher(this, sequence, filteredResults, null);
720         //            matcher.AddAllPartialMatchedToResults(firstResults);
721         //            return filteredResults;
722         //        }
723 
724         //        return new HashSet<ulong>();
725         //    });
726         //}
```

```
727
728         [MethodImpl(MethodImplOptions.AggressiveInlining)]
729         public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
730         {
731             return _sync.ExecuteReadOperation((Func<HashSet<ulong>>)(() =>
732             {
733                 if (sequence.Length > 0)
734                 {
735                     ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links,
                        ↪  (IList<ulong>)sequence);
736                     var firstResults = new HashSet<ulong>();
737                     var lastResults = new HashSet<ulong>();
738                     var first = sequence.First(x => x != Constants.Any);
739                     var last = sequence.Last(x => x != Constants.Any);
740                     AllUsagesCore(first, firstResults);
741                     AllUsagesCore(last, lastResults);
742                     firstResults.IntersectWith(lastResults);
743                     //for (var i = 0; i < sequence.Length; i++)
744                     //    AllUsagesCore(sequence[i], results);
745                     var filteredResults = new HashSet<ulong>();
746                     var matcher = new Matcher(this, sequence, filteredResults, null);
747                     matcher.AddAllPartialMatchedToResults(firstResults);
748                     return filteredResults;
749                 }
750                 return new HashSet<ulong>();
751             }));
752         }
753
754         [MethodImpl(MethodImplOptions.AggressiveInlining)]
755         public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
            ↪  IList<ulong> sequence)
756         {
757             return _sync.ExecuteReadOperation(() =>
758             {
759                 if (sequence.Count > 0)
760                 {
761                     Links.EnsureLinkExists(sequence);
762                     var results = new HashSet<LinkIndex>();
763                     //var nextResults = new HashSet<ulong>();
764                     //for (var i = 0; i < sequence.Length; i++)
765                     //{
766                     //    AllUsagesCore(sequence[i], nextResults);
767                     //    if (results.IsNullOrEmpty())
768                     //    {
769                     //        results = nextResults;
770                     //        nextResults = new HashSet<ulong>();
771                     //    }
772                     //    else
773                     //    {
774                     //        results.IntersectWith(nextResults);
775                     //        nextResults.Clear();
776                     //    }
777                     //}
778                     var collector1 = new AllUsagesCollector1(Links.Unsync, results);
779                     collector1.Collect(Links.Unsync.GetLink(sequence[0]));
780                     var next = new HashSet<ulong>();
781                     for (var i = 1; i < sequence.Count; i++)
782                     {
783                         var collector = new AllUsagesCollector1(Links.Unsync, next);
784                         collector.Collect(Links.Unsync.GetLink(sequence[i]));
785
786                         results.IntersectWith(next);
787                         next.Clear();
788                     }
789                     var filteredResults = new HashSet<ulong>();
790                     var matcher = new Matcher(this, sequence, filteredResults, null,
                        ↪  readAsElements);
791                     matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
                        ↪  x)); // OrderBy is a Hack
792                     return filteredResults;
793                 }
794                 return new HashSet<ulong>();
795             });
796         }
797
798         // Does not work
799         //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
            ↪  params ulong[] sequence)
```

```
800         //{
801         //    var visited = new HashSet<ulong>();
802         //    var results = new HashSet<ulong>();
803         //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
            ↪  true; }, readAsElements);
804         //    var last = sequence.Length - 1;
805         //    for (var i = 0; i < last; i++)
806         //    {
807         //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
808         //    }
809         //    return results;
810         //}
811
812         [MethodImpl(MethodImplOptions.AggressiveInlining)]
813         public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
814         {
815             return _sync.ExecuteReadOperation(() =>
816             {
817                 if (sequence.Length > 0)
818                 {
819                     Links.EnsureLinkExists(sequence);
820                     //var firstElement = sequence[0];
821                     //if (sequence.Length == 1)
822                     //{
823                     //    //results.Add(firstElement);
824                     //    return results;
825                     //}
826                     //if (sequence.Length == 2)
827                     //{
828                     //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
829                     //    //if (doublet != Doublets.Links.Null)
830                     //    //    results.Add(doublet);
831                     //    return results;
832                     //}
833                     //var lastElement = sequence[sequence.Length - 1];
834                     //Func<ulong, bool> handler = x =>
835                     //{
836                     //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
                        ↪  results.Add(x);
837                     //    return true;
838                     //};
839                     //if (sequence.Length >= 2)
840                     //    StepRight(handler, sequence[0], sequence[1]);
841                     //var last = sequence.Length - 2;
842                     //for (var i = 1; i < last; i++)
843                     //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
844                     //if (sequence.Length >= 3)
845                     //    StepLeft(handler, sequence[sequence.Length - 2],
                        ↪  sequence[sequence.Length - 1]);
846                     //////if (sequence.Length == 1)
847                     //////{
848                     //////    throw new NotImplementedException(); // all sequences, containing
                        ↪  this element?
849                     //////}
850                     //////if (sequence.Length == 2)
851                     //////{
852                     //////    var results = new List<ulong>();
853                     //////    PartialStepRight(results.Add, sequence[0], sequence[1]);
854                     //////    return results;
855                     //////}
856                     //////var matches = new List<List<ulong>>();
857                     //////var last = sequence.Length - 1;
858                     //////for (var i = 0; i < last; i++)
859                     //////{
860                     //////    var results = new List<ulong>();
861                     //////    //StepRight(results.Add, sequence[i], sequence[i + 1]);
862                     //////    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
863                     //////    if (results.Count > 0)
864                     //////        matches.Add(results);
865                     //////    else
866                     //////        return results;
867                     //////    if (matches.Count == 2)
868                     //////    {
869                     //////        var merged = new List<ulong>();
870                     //////        for (var j = 0; j < matches[0].Count; j++)
871                     //////            for (var k = 0; k < matches[1].Count; k++)
```

```
872                     //////                CloseInnerConnections(merged.Add, matches[0][j],
                        ↪  matches[1][k]);
873                     //////        if (merged.Count > 0)
874                     //////            matches = new List<List<ulong>> { merged };
875                     //////        else
876                     //////            return new List<ulong>();
877                     //////    }
878                     //////}
879                     //////if (matches.Count > 0)
880                     //////{
881                     //////    var usages = new HashSet<ulong>();
882                     //////    for (int i = 0; i < sequence.Length; i++)
883                     //////    {
884                     //////        AllUsagesCore(sequence[i], usages);
885                     //////    }
886                     //////    //for (int i = 0; i < matches[0].Count; i++)
887                     //////    //    AllUsagesCore(matches[0][i], usages);
888                     //////    //usages.UnionWith(matches[0]);
889                     //////    return usages.ToList();
890                     //////}
891                     var firstLinkUsages = new HashSet<ulong>();
892                     AllUsagesCore(sequence[0], firstLinkUsages);
893                     firstLinkUsages.Add(sequence[0]);
894                     //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
                        ↪  sequence[0] }; // or all sequences, containing this element?
895                     //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
                        ↪  1).ToList();
896                     var results = new HashSet<ulong>();
897                     foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
                        ↪  firstLinkUsages, 1))
898                     {
899                         AllUsagesCore(match, results);
900                     }
901                     return results.ToList();
902                 }
903                 return new List<ulong>();
904             });
905         }
906 
907         /// <remarks>
908         /// TODO: Может потробоваться ограничение на уровень глубины рекурсии
909         /// </remarks>
910         [MethodImpl(MethodImplOptions.AggressiveInlining)]
911         public HashSet<ulong> AllUsages(ulong link)
912         {
913             return _sync.ExecuteReadOperation(() =>
914             {
915                 var usages = new HashSet<ulong>();
916                 AllUsagesCore(link, usages);
917                 return usages;
918             });
919         }
920 
921         // При сборе всех использований (последовательностей) можно сохранять обратный путь к
                ↪  той связи с которой начинался поиск (STTTSSSTT),
922         // причём достаточно одного бита для хранения перехода влево или вправо
923         [MethodImpl(MethodImplOptions.AggressiveInlining)]
924         private void AllUsagesCore(ulong link, HashSet<ulong> usages)
925         {
926             bool handler(ulong doublet)
927             {
928                 if (usages.Add(doublet))
929                 {
930                     AllUsagesCore(doublet, usages);
931                 }
932                 return true;
933             }
934             Links.Unsync.Each(link, Constants.Any, handler);
935             Links.Unsync.Each(Constants.Any, link, handler);
936         }
937 
938         [MethodImpl(MethodImplOptions.AggressiveInlining)]
939         public HashSet<ulong> AllBottomUsages(ulong link)
940         {
941             return _sync.ExecuteReadOperation(() =>
942             {
943                 var visits = new HashSet<ulong>();
```

```
            var usages = new HashSet<ulong>();
            AllBottomUsagesCore(link, visits, usages);
            return usages;
        });
    }

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

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

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

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

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public void CalculateAllUsages2(ulong[] totals)
```

```
        {
            var calculator = new AllUsagesCalculator2(Links, totals);
            calculator.Calculate();
        }

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
            ↪  CalculateCore);

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

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
            ↪  CalculateCore);

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool CalculateCore(ulong link)
            {
                // TODO: Проработать защиту от зацикливания
                // Основано на SequenceWalker.WalkLeft
                Func<ulong, ulong> getSource = _links.Unsync.GetSource;
                Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
                Func<ulong, bool> isElement = IsElement;
                void visitLeaf(ulong parent)
                {
```

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
            {
                _links = links;
                _usages = usages;
            }

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

```

```
1173            private class AllUsagesCollector1
1174            {
1175                private readonly ILinks<ulong> _links;
1176                private readonly HashSet<ulong> _usages;
1177                private readonly ulong _continue;
1178
1179                [MethodImpl(MethodImplOptions.AggressiveInlining)]
1180                public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1181                {
1182                    _links = links;
1183                    _usages = usages;
1184                    _continue = _links.Constants.Continue;
1185                }
1186
1187                [MethodImpl(MethodImplOptions.AggressiveInlining)]
1188                public ulong Collect(IList<ulong> link)
1189                {
1190                    var linkIndex = _links.GetIndex(link);
1191                    if (_usages.Add(linkIndex))
1192                    {
1193                        _links.Each(Collect, _links.Constants.Any, linkIndex);
1194                    }
1195                    return _continue;
1196                }
1197            }
1198
1199            private class AllUsagesCollector2
1200            {
1201                private readonly ILinks<ulong> _links;
1202                private readonly BitString _usages;
1203
1204                [MethodImpl(MethodImplOptions.AggressiveInlining)]
1205                public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1206                {
1207                    _links = links;
1208                    _usages = usages;
1209                }
1210
1211                [MethodImpl(MethodImplOptions.AggressiveInlining)]
1212                public bool Collect(ulong link)
1213                {
1214                    if (_usages.Add((long)link))
1215                    {
1216                        _links.Each(link, _links.Constants.Any, Collect);
1217                        _links.Each(_links.Constants.Any, link, Collect);
1218                    }
1219                    return true;
1220                }
1221            }
1222
1223            private class AllUsagesIntersectingCollector
1224            {
1225                private readonly SynchronizedLinks<ulong> _links;
1226                private readonly HashSet<ulong> _intersectWith;
1227                private readonly HashSet<ulong> _usages;
1228                private readonly HashSet<ulong> _enter;
1229
1230                [MethodImpl(MethodImplOptions.AggressiveInlining)]
1231                public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
                    ↪ intersectWith, HashSet<ulong> usages)
1232                {
1233                    _links = links;
1234                    _intersectWith = intersectWith;
1235                    _usages = usages;
1236                    _enter = new HashSet<ulong>(); // защита от зацикливания
1237                }
1238
1239                [MethodImpl(MethodImplOptions.AggressiveInlining)]
1240                public bool Collect(ulong link)
1241                {
1242                    if (_enter.Add(link))
1243                    {
1244                        if (_intersectWith.Contains(link))
1245                        {
1246                            _usages.Add(link);
1247                        }
1248                        _links.Unsync.Each(link, _links.Constants.Any, Collect);
1249                        _links.Unsync.Each(_links.Constants.Any, link, Collect);
1250                    }
1251                    return true;
```

```
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
        ↪  right)
        {
            TryStepLeftUp(handler, left, right);
            TryStepRightUp(handler, right, left);
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
        ↪  links, params ulong[] sequence)
        {
            if (sequence == null)
```

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new HashSet<ulong>();
```

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new HashSet<ulong>();
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureLinkExists(linksToConnect);
                    var collector1 = new AllUsagesCollector(Links, results);
                    collector1.Collect(linksToConnect[0]);
                    //AllUsagesCore(linksToConnect[0], results);
                    for (var i = 1; i < linksToConnect.Length; i++)
                    {
                        var next = new HashSet<ulong>();
                        var collector = new AllUsagesIntersectingCollector(Links, results, next);
                        collector.Collect(linksToConnect[i]);
                        //AllUsagesCore(linksToConnect[i], next);
                        //results.IntersectWith(next);
                        results = next;
                    }
                }
                return results;
            });
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static ulong[] Simplify(ulong[] sequence)
        {
            // Считаем новый размер последовательности
            long newLength = 0;
            var zeroOrManyStepped = false;
            for (var i = 0; i < sequence.Length; i++)
            {
                if (sequence[i] == ZeroOrMany)
```

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> GetSimilarSequences() => new List<ulong>();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Prediction()
        {
            //_links
            //sequences
        }

        #region From Triplets

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> CollectMatchingSequences(ulong[] links)
        {
            if (links.Length == 1)
            {
                throw new InvalidOperationException("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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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);
                        }
                    }
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            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();
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
            ↪  base.IsElement(link);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            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);
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            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
                    };
                }
            }
        }
        if (patternBlock.Type != PatternBlockType.Undefined)
        {
            pattern.Add(patternBlock);
        }
        return pattern;
    }

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

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

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

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

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

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private bool PatternMatchCore(LinkIndex element)
    {
        if (_patternPosition >= _pattern.Count)
        {
            _patternPosition = -2;
            return false;
```

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

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

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

        #endregion
    }
}
```

1.94 ./csharp/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.
```

```
43      ///
44      /// Можно ли как-то сделать один общий интерфейс
45      ///
46      ///
47      /// Блокчейн и/или гит для распределённой записи транзакций.
48      ///
49      /// </remarks>
50      public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
        ↪  (после завершения реализации Sequences)
51      {
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          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
63          {
64              Links = links;
65              _sync = links.SyncRoot;
66              Options = options;
67              Options.ValidateOptions();
68              Options.InitOptions(Links);
69              Constants = links.Constants;
70          }
71
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
            ↪  SequencesOptions<LinkIndex>()) { }
74
75          [MethodImpl(MethodImplOptions.AggressiveInlining)]
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          [MethodImpl(MethodImplOptions.AggressiveInlining)]
99          private LinkIndex GetSequenceElements(LinkIndex sequence)
100         {
101             if (Options.UseSequenceMarker)
102             {
103                 var linkContents = new Link<ulong>(Links.GetLink(sequence));
104                 if (linkContents.Source == Options.SequenceMarkerLink)
105                 {
106                     return linkContents.Target;
107                 }
108                 if (linkContents.Target == Options.SequenceMarkerLink)
109                 {
110                     return linkContents.Source;
111                 }
112             }
113             return sequence;
114         }
115
116         #region Count
117
118         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

```
197             {
198                 Options.Index.Add(sequence);
199             }
200             var sequenceRoot = default(LinkIndex);
201             if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
202             {
203                 var matches = Each(restrictions);
204                 if (matches.Count > 0)
205                 {
206                     sequenceRoot = matches[0];
207                 }
208             }
209             else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
210             {
211                 return CompactCore(sequence);
212             }
213             if (sequenceRoot == default)
214             {
215                 sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
216             }
217             if (Options.UseSequenceMarker)
218             {
219                 return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
220             }
221             return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
222         }
223 
224         #endregion
225 
226         #region Each
227 
228         [MethodImpl(MethodImplOptions.AggressiveInlining)]
229         public List<LinkIndex> Each(IList<LinkIndex> sequence)
230         {
231             var results = new List<LinkIndex>();
232             var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
233             Each(filler.AddFirstAndReturnConstant, sequence);
234             return results;
235         }
236 
237         [MethodImpl(MethodImplOptions.AggressiveInlining)]
238         public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
            ↪ restrictions)
239         {
240             return _sync.ExecuteReadOperation(() =>
241             {
242                 if (restrictions.IsNullOrEmpty())
243                 {
244                     return Constants.Continue;
245                 }
246                 Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
247                 if (restrictions.Count == 1)
248                 {
249                     var link = restrictions[0];
250                     var any = Constants.Any;
251                     if (link == any)
252                     {
253                         if (Options.UseSequenceMarker)
254                         {
255                             return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
                                ↪ Options.SequenceMarkerLink, any));
256                         }
257                         else
258                         {
259                             return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
                                ↪ any));
260                         }
261                     }
262                     if (Options.UseSequenceMarker)
263                     {
264                         var sequenceLinkValues = Links.Unsync.GetLink(link);
265                         if (sequenceLinkValues[Constants.SourcePart] ==
                            ↪ Options.SequenceMarkerLink)
266                         {
267                             link = sequenceLinkValues[Constants.TargetPart];
268                         }
269                     }
270                     var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
```

```
                sequence[0] = link;
                return handler(sequence);
            }
            else if (restrictions.Count == 2)
            {
                throw new NotImplementedException();
            }
            else if (restrictions.Count == 3)
            {
                return Links.Unsync.Each(handler, restrictions);
            }
            else
            {
                var sequence = restrictions.SkipFirst();
                if (Options.UseIndex && !Options.Index.MightContain(sequence))
                {
                    return Constants.Break;
                }
                return EachCore(handler, sequence);
            }
        });
    }

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

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

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
342         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));
343
344         [MethodImpl(MethodImplOptions.AggressiveInlining)]
345         private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
            ↪ right, LinkIndex stepFrom)
346         {
347             var upStep = stepFrom;
348             var firstSource = Links.Unsync.GetTarget(upStep);
349             while (firstSource != right && firstSource != upStep)
350             {
351                 upStep = firstSource;
352                 firstSource = Links.Unsync.GetSource(upStep);
353             }
354             if (firstSource == right)
355             {
356                 return handler(new LinkAddress<LinkIndex>(stepFrom));
357             }
358             return Constants.Continue;
359         }
360
361         [MethodImpl(MethodImplOptions.AggressiveInlining)]
362         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));
363
364         [MethodImpl(MethodImplOptions.AggressiveInlining)]
365         private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
            ↪ left, LinkIndex stepFrom)
366         {
367             var upStep = stepFrom;
368             var firstTarget = Links.Unsync.GetSource(upStep);
369             while (firstTarget != left && firstTarget != upStep)
370             {
371                 upStep = firstTarget;
372                 firstTarget = Links.Unsync.GetTarget(upStep);
373             }
374             if (firstTarget == left)
375             {
376                 return handler(new LinkAddress<LinkIndex>(stepFrom));
377             }
378             return Constants.Continue;
379         }
380
381         #endregion
382
383         #region Update
384
385         [MethodImpl(MethodImplOptions.AggressiveInlining)]
386         public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
387         {
388             var sequence = restrictions.SkipFirst();
389             var newSequence = substitution.SkipFirst();
390             if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
391             {
392                 return Constants.Null;
393             }
394             if (sequence.IsNullOrEmpty())
395             {
396                 return Create(substitution);
397             }
398             if (newSequence.IsNullOrEmpty())
399             {
400                 Delete(restrictions);
401                 return Constants.Null;
402             }
403             return _sync.ExecuteWriteOperation((Func<ulong>)(() =>
404             {
405                 ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
406                 Links.EnsureLinkExists(newSequence);
407                 return UpdateCore(sequence, newSequence);
408             }));
409         }
410
411         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
412         private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence)
413         {
414             LinkIndex bestVariant;
415             if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
                ↪  !sequence.EqualTo(newSequence))
416             {
417                 bestVariant = CompactCore(newSequence);
418             }
419             else
420             {
421                 bestVariant = CreateCore(newSequence);
422             }
423             // TODO: Check all options only ones before loop execution
424             // Возможно нужно две версии Each, возвращающий фактические последовательности и с
                ↪  маркером,
425             // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
                ↪  можно получить имея только фактические последовательности.
426             foreach (var variant in Each(sequence))
427             {
428                 if (variant != bestVariant)
429                 {
430                     UpdateOneCore(variant, bestVariant);
431                 }
432             }
433             return bestVariant;
434         }
435 
436         [MethodImpl(MethodImplOptions.AggressiveInlining)]
437         private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
438         {
439             if (Options.UseGarbageCollection)
440             {
441                 var sequenceElements = GetSequenceElements(sequence);
442                 var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
443                 var sequenceLink = GetSequenceByElements(sequenceElements);
444                 var newSequenceElements = GetSequenceElements(newSequence);
445                 var newSequenceLink = GetSequenceByElements(newSequenceElements);
446                 if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
447                 {
448                     if (sequenceLink != Constants.Null)
449                     {
450                         Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
451                     }
452                     Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
453                 }
454                 ClearGarbage(sequenceElementsContents.Source);
455                 ClearGarbage(sequenceElementsContents.Target);
456             }
457             else
458             {
459                 if (Options.UseSequenceMarker)
460                 {
461                     var sequenceElements = GetSequenceElements(sequence);
462                     var sequenceLink = GetSequenceByElements(sequenceElements);
463                     var newSequenceElements = GetSequenceElements(newSequence);
464                     var newSequenceLink = GetSequenceByElements(newSequenceElements);
465                     if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
466                     {
467                         if (sequenceLink != Constants.Null)
468                         {
469                             Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
470                         }
471                         Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
472                     }
473                 }
474                 else
475                 {
476                     if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
477                     {
478                         Links.Unsync.MergeAndDelete(sequence, newSequence);
479                     }
480                 }
481             }
482         }
483 
484         #endregion
485 
486         #region Delete
```

```
487
488         [MethodImpl(MethodImplOptions.AggressiveInlining)]
489         public void Delete(IList<LinkIndex> restrictions)
490         {
491             _sync.ExecuteWriteOperation(() =>
492             {
493                 var sequence = restrictions.SkipFirst();
494                 // TODO: Check all options only ones before loop execution
495                 foreach (var linkToDelete in Each(sequence))
496                 {
497                     DeleteOneCore(linkToDelete);
498                 }
499             });
500         }
501
502         [MethodImpl(MethodImplOptions.AggressiveInlining)]
503         private void DeleteOneCore(LinkIndex link)
504         {
505             if (Options.UseGarbageCollection)
506             {
507                 var sequenceElements = GetSequenceElements(link);
508                 var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
509                 var sequenceLink = GetSequenceByElements(sequenceElements);
510                 if (Options.UseCascadeDelete || CountUsages(link) == 0)
511                 {
512                     if (sequenceLink != Constants.Null)
513                     {
514                         Links.Unsync.Delete(sequenceLink);
515                     }
516                     Links.Unsync.Delete(link);
517                 }
518                 ClearGarbage(sequenceElementsContents.Source);
519                 ClearGarbage(sequenceElementsContents.Target);
520             }
521             else
522             {
523                 if (Options.UseSequenceMarker)
524                 {
525                     var sequenceElements = GetSequenceElements(link);
526                     var sequenceLink = GetSequenceByElements(sequenceElements);
527                     if (Options.UseCascadeDelete || CountUsages(link) == 0)
528                     {
529                         if (sequenceLink != Constants.Null)
530                         {
531                             Links.Unsync.Delete(sequenceLink);
532                         }
533                         Links.Unsync.Delete(link);
534                     }
535                 }
536                 else
537                 {
538                     if (Options.UseCascadeDelete || CountUsages(link) == 0)
539                     {
540                         Links.Unsync.Delete(link);
541                     }
542                 }
543             }
544         }
545
546         #endregion
547
548         #region Compactification
549
550         [MethodImpl(MethodImplOptions.AggressiveInlining)]
551         public void CompactAll()
552         {
553             _sync.ExecuteWriteOperation(() =>
554             {
555                 var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
556                 for (int i = 0; i < sequences.Count; i++)
557                 {
558                     var sequence = this.ToList(sequences[i]);
559                     Compact(sequence.ShiftRight());
560                 }
561             });
562         }
563
564         /// <remarks>
```

```
565             /// bestVariant можно выбирать по максимальному числу использований,
566             /// но балансированный позволяет гарантировать уникальность (если есть возможность,
567             /// гарантировать его использование в других местах).
568             ///
569             /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
570             /// </remarks>
571             [MethodImpl(MethodImplOptions.AggressiveInlining)]
572             public LinkIndex Compact(IList<LinkIndex> sequence)
573             {
574                 return _sync.ExecuteWriteOperation(() =>
575                 {
576                     if (sequence.IsNullOrEmpty())
577                     {
578                         return Constants.Null;
579                     }
580                     Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
581                     return CompactCore(sequence);
582                 });
583             }
584 
585             [MethodImpl(MethodImplOptions.AggressiveInlining)]
586             private LinkIndex CompactCore(IList<LinkIndex> sequence) => UpdateCore(sequence,
                 ↪ sequence);
587 
588             #endregion
589 
590             #region Garbage Collection
591 
592             /// <remarks>
593             /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
                 ↪ определить извне или в унаследованном классе
594             /// </remarks>
595             [MethodImpl(MethodImplOptions.AggressiveInlining)]
596             private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
                 ↪ !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
597 
598             [MethodImpl(MethodImplOptions.AggressiveInlining)]
599             private void ClearGarbage(LinkIndex link)
600             {
601                 if (IsGarbage(link))
602                 {
603                     var contents = new Link<ulong>(Links.GetLink(link));
604                     Links.Unsync.Delete(link);
605                     ClearGarbage(contents.Source);
606                     ClearGarbage(contents.Target);
607                 }
608             }
609 
610             #endregion
611 
612             #region Walkers
613 
614             [MethodImpl(MethodImplOptions.AggressiveInlining)]
615             public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
616             {
617                 return _sync.ExecuteReadOperation(() =>
618                 {
619                     var links = Links.Unsync;
620                     foreach (var part in Options.Walker.Walk(sequence))
621                     {
622                         if (!handler(part))
623                         {
624                             return false;
625                         }
626                     }
627                     return true;
628                 });
629             }
630 
631             public class Matcher : RightSequenceWalker<LinkIndex>
632             {
633                 private readonly Sequences _sequences;
634                 private readonly IList<LinkIndex> _patternSequence;
635                 private readonly HashSet<LinkIndex> _linksInSequence;
636                 private readonly HashSet<LinkIndex> _results;
637                 private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
638                 private readonly HashSet<LinkIndex> _readAsElements;
639                 private int _filterPosition;
640 
641                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
642             public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
                ↪  HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
                ↪  HashSet<LinkIndex> readAsElements = null)
643                 : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
644             {
645                 _sequences = sequences;
646                 _patternSequence = patternSequence;
647                 _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                    ↪  _links.Constants.Any && x != ZeroOrMany));
648                 _results = results;
649                 _stopableHandler = stopableHandler;
650                 _readAsElements = readAsElements;
651             }
652
653             [MethodImpl(MethodImplOptions.AggressiveInlining)]
654             protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
                ↪  (_readAsElements != null && _readAsElements.Contains(link)) ||
                ↪  _linksInSequence.Contains(link);
655
656             [MethodImpl(MethodImplOptions.AggressiveInlining)]
657             public bool FullMatch(LinkIndex sequenceToMatch)
658             {
659                 _filterPosition = 0;
660                 foreach (var part in Walk(sequenceToMatch))
661                 {
662                     if (!FullMatchCore(part))
663                     {
664                         break;
665                     }
666                 }
667                 return _filterPosition == _patternSequence.Count;
668             }
669
670             [MethodImpl(MethodImplOptions.AggressiveInlining)]
671             private bool FullMatchCore(LinkIndex element)
672             {
673                 if (_filterPosition == _patternSequence.Count)
674                 {
675                     _filterPosition = -2; // Длиннее чем нужно
676                     return false;
677                 }
678                 if (_patternSequence[_filterPosition] != _links.Constants.Any
679                  && element != _patternSequence[_filterPosition])
680                 {
681                     _filterPosition = -1;
682                     return false; // Начинается/Продолжается иначе
683                 }
684                 _filterPosition++;
685                 return true;
686             }
687
688             [MethodImpl(MethodImplOptions.AggressiveInlining)]
689             public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
690             {
691                 var sequenceToMatch = restrictions[_links.Constants.IndexPart];
692                 if (FullMatch(sequenceToMatch))
693                 {
694                     _results.Add(sequenceToMatch);
695                 }
696             }
697
698             [MethodImpl(MethodImplOptions.AggressiveInlining)]
699             public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
700             {
701                 var sequenceToMatch = restrictions[_links.Constants.IndexPart];
702                 if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
703                 {
704                     return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
705                 }
706                 return _links.Constants.Continue;
707             }
708
709             [MethodImpl(MethodImplOptions.AggressiveInlining)]
710             public LinkIndex HandleFullMatchedSequence(IList<LinkIndex> restrictions)
711             {
712                 var sequenceToMatch = restrictions[_links.Constants.IndexPart];
713                 var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
714                 if (sequence != _links.Constants.Null && FullMatch(sequenceToMatch) &&
                    ↪  _results.Add(sequenceToMatch))
```

```
715                 {
716                     return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
717                 }
718                 return _links.Constants.Continue;
719             }
720 
721             /// <remarks>
722             /// TODO: Add support for LinksConstants.Any
723             /// </remarks>
724             [MethodImpl(MethodImplOptions.AggressiveInlining)]
725             public bool PartialMatch(LinkIndex sequenceToMatch)
726             {
727                 _filterPosition = -1;
728                 foreach (var part in Walk(sequenceToMatch))
729                 {
730                     if (!PartialMatchCore(part))
731                     {
732                         break;
733                     }
734                 }
735                 return _filterPosition == _patternSequence.Count - 1;
736             }
737 
738             [MethodImpl(MethodImplOptions.AggressiveInlining)]
739             private bool PartialMatchCore(LinkIndex element)
740             {
741                 if (_filterPosition == (_patternSequence.Count - 1))
742                 {
743                     return false; // Нашлось
744                 }
745                 if (_filterPosition >= 0)
746                 {
747                     if (element == _patternSequence[_filterPosition + 1])
748                     {
749                         _filterPosition++;
750                     }
751                     else
752                     {
753                         _filterPosition = -1;
754                     }
755                 }
756                 if (_filterPosition < 0)
757                 {
758                     if (element == _patternSequence[0])
759                     {
760                         _filterPosition = 0;
761                     }
762                 }
763                 return true; // Ищем дальше
764             }
765 
766             [MethodImpl(MethodImplOptions.AggressiveInlining)]
767             public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
768             {
769                 if (PartialMatch(sequenceToMatch))
770                 {
771                     _results.Add(sequenceToMatch);
772                 }
773             }
774 
775             [MethodImpl(MethodImplOptions.AggressiveInlining)]
776             public LinkIndex HandlePartialMatched(IList<LinkIndex> restrictions)
777             {
778                 var sequenceToMatch = restrictions[_links.Constants.IndexPart];
779                 if (PartialMatch(sequenceToMatch))
780                 {
781                     return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
782                 }
783                 return _links.Constants.Continue;
784             }
785 
786             [MethodImpl(MethodImplOptions.AggressiveInlining)]
787             public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)
788             {
789                 foreach (var sequenceToMatch in sequencesToMatch)
790                 {
791                     if (PartialMatch(sequenceToMatch))
792                     {
793                         _results.Add(sequenceToMatch);
```

```
                    }
                }
            }

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

        #endregion
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
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
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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());
        }

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

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

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

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

namespace Platform.Data.Doublets.Sequences
{
    public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
    ↪ ILinks<TLink> must contain GetConstants function.
```

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

        public TLink SequenceMarkerLink
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool UseCascadeUpdate
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool UseCascadeDelete
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool UseIndex
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        } // TODO: Update Index on sequence update/delete.

        public bool UseSequenceMarker
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool UseCompression
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool UseGarbageCollection
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public bool EnforceSingleSequenceVersionOnWriteBasedOnNew
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
99              set;
100         }
101
102         public IConverter<IList<TLink>, TLink> LinksToSequenceConverter
103         {
104             [MethodImpl(MethodImplOptions.AggressiveInlining)]
105             get;
106             [MethodImpl(MethodImplOptions.AggressiveInlining)]
107             set;
108         }
109
110         public ISequenceIndex<TLink> Index
111         {
112             [MethodImpl(MethodImplOptions.AggressiveInlining)]
113             get;
114             [MethodImpl(MethodImplOptions.AggressiveInlining)]
115             set;
116         }
117
118         public ISequenceWalker<TLink> Walker
119         {
120             [MethodImpl(MethodImplOptions.AggressiveInlining)]
121             get;
122             [MethodImpl(MethodImplOptions.AggressiveInlining)]
123             set;
124         }
125
126         public bool ReadFullSequence
127         {
128             [MethodImpl(MethodImplOptions.AggressiveInlining)]
129             get;
130             [MethodImpl(MethodImplOptions.AggressiveInlining)]
131             set;
132         }
133
134         // TODO: Реализовать компактификацию при чтении
135         //public bool EnforceSingleSequenceVersionOnRead { get; set; }
136         //public bool UseRequestMarker { get; set; }
137         //public bool StoreRequestResults { get; set; }
138
139         [MethodImpl(MethodImplOptions.AggressiveInlining)]
140         public void InitOptions(ISynchronizedLinks<TLink> links)
141         {
142             if (UseSequenceMarker)
143             {
144                 if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
145                 {
146                     SequenceMarkerLink = links.CreatePoint();
147                 }
148                 else
149                 {
150                     if (!links.Exists(SequenceMarkerLink))
151                     {
152                         var link = links.CreatePoint();
153                         if (!_equalityComparer.Equals(link, SequenceMarkerLink))
154                         {
155                             throw new InvalidOperationException("Cannot recreate sequence marker
                                 ↪ link.");
156                         }
157                     }
158                 }
159                 if (MarkedSequenceMatcher == null)
160                 {
161                     MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
                         ↪ SequenceMarkerLink);
162                 }
163             }
164             var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
165             if (UseCompression)
166             {
167                 if (LinksToSequenceConverter == null)
168                 {
169                     ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
170                     if (UseSequenceMarker)
171                     {
172                         totalSequenceSymbolFrequencyCounter = new
                             ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
                             ↪ MarkedSequenceMatcher);
173                     }
```

```
                    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>());
            }
        }

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

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

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

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

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

1.98 ./csharp/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>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪  isElement) : base(links, stack, isElement) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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 links = _links;
            var parts = links.GetLink(element);
            var start = links.Constants.SourcePart;
            for (var i = parts.Count - 1; i >= start; i--)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
    }
}
```

1.99 ./csharp/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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
        ↪ base(links) => _isElement = isElement;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
        ↪ _links.IsPartialPoint;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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 links = _links;
                        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;
        }
    }
}
```

1.100 ./csharp/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>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪  isElement) : base(links, stack, isElement) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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.SourcePart; i < parts.Count; i++)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
    }
}
```

1.101 ./csharp/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;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
        ↪  stack, links.IsPartialPoint) { }

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

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

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

        private readonly TLink _stack;

        public bool IsEmpty
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Peek(), _stack);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Stack(ILinks<TLink> links, TLink stack) : base(links) => _stack = stack;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink GetStackMarker() => _links.GetSource(_stack);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink GetTop() => _links.GetTarget(_stack);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Peek() => _links.GetTarget(GetTop());

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
        ↪  _links.GetOrCreate(GetTop(), element));
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Stacks
{
    public static class StackExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink CreateStack<TLink>(this ILinks<TLink> links, TLink stackMarker)
        {
            var stackPoint = links.CreatePoint();
            var stack = links.Update(stackPoint, stackMarker, stackPoint);
            return stack;
        }
    }
}
```

1.104 ./csharp/Platform.Data.Doublets/SynchronizedLinks.cs

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

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

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

        public ISynchronization SyncRoot
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public ILinks<TLinkAddress> Sync
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public ILinks<TLinkAddress> Unsync
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
        ↪  ReaderWriterLockSynchronization(), links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
        {
```

```
48              SyncRoot = synchronization;
49              Sync = this;
50              Unsync = links;
51              Constants = links.Constants;
52          }
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          public TLinkAddress Count(IList<TLinkAddress> restriction) =>
            ↪  SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
            ↪  IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
            ↪  restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
59
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
            ↪  SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
            ↪  substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
            ↪  Unsync.Update);
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          public void Delete(IList<TLinkAddress> restrictions) =>
            ↪  SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
68
69          //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
            ↪  IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
70          //{
71          //    if (restriction != null && substitution != null &&
            ↪  !substitution.EqualTo(restriction))
72          //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
            ↪  substitution, substitutedHandler, Unsync.Trigger);
73
74          //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
            ↪  substitutedHandler, Unsync.Trigger);
75          //}
76      }
77  }
```

1.105 ./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs

```
1   using System;
2   using System.Text;
3   using System.Collections.Generic;
4   using System.Runtime.CompilerServices;
5   using Platform.Singletons;
6   using Platform.Data.Doublets.Unicode;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets
11  {
12      public static class UInt64LinksExtensions
13      {
14          public static readonly LinksConstants<ulong> Constants =
            ↪  Default<LinksConstants<ulong>>.Instance;
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
21          {
22              if (sequence == null)
23              {
24                  return false;
25              }
26              var constants = links.Constants;
27              for (var i = 0; i < sequence.Length; i++)
28              {
29                  if (sequence[i] == constants.Any)
30                  {
31                      return true;
32                  }
33              }
34              return false;
35          }
```

```
36
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
            ↪  Func<Link<ulong>, bool> isElement, bool renderIndex = false, bool renderDebug =
            ↪  false)
39          {
40              var sb = new StringBuilder();
41              var visited = new HashSet<ulong>();
42              links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
                ↪  innerSb.Append(link.Index), renderIndex, renderDebug);
43              return sb.ToString();
44          }
45
46          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47          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)
48          {
49              var sb = new StringBuilder();
50              var visited = new HashSet<ulong>();
51              links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
                ↪  renderDebug);
52              return sb.ToString();
53          }
54
55          [MethodImpl(MethodImplOptions.AggressiveInlining)]
56          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)
57          {
58              if (sb == null)
59              {
60                  throw new ArgumentNullException(nameof(sb));
61              }
62              if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
                ↪  Constants.Itself)
63              {
64                  return;
65              }
66              if (links.Exists(linkIndex))
67              {
68                  if (visited.Add(linkIndex))
69                  {
70                      sb.Append('(');
71                      var link = new Link<ulong>(links.GetLink(linkIndex));
72                      if (renderIndex)
73                      {
74                          sb.Append(link.Index);
75                          sb.Append(':');
76                      }
77                      if (link.Source == link.Index)
78                      {
79                          sb.Append(link.Index);
80                      }
81                      else
82                      {
83                          var source = new Link<ulong>(links.GetLink(link.Source));
84                          if (isElement(source))
85                          {
86                              appendElement(sb, source);
87                          }
88                          else
89                          {
90                              links.AppendStructure(sb, visited, source.Index, isElement,
                                ↪  appendElement, renderIndex);
91                          }
92                      }
93                      sb.Append(' ');
94                      if (link.Target == link.Index)
95                      {
96                          sb.Append(link.Index);
97                      }
98                      else
99                      {
100                         var target = new Link<ulong>(links.GetLink(link.Target));
101                         if (isElement(target))
102                         {
```

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

1.106 ./csharp/Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

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

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

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

```
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              [MethodImpl(MethodImplOptions.AggressiveInlining)]
95              public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
                ↪  transactionId, Link<ulong> before, Link<ulong> after)
96              {
97                  TransactionId = transactionId;
98                  Before = before;
99                  After = after;
100                 Timestamp = uniqueTimestampFactory.Create();
101             }
102
103             [MethodImpl(MethodImplOptions.AggressiveInlining)]
104             public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
                ↪  transactionId, Link<ulong> before) : this(uniqueTimestampFactory, transactionId,
                ↪  before, default) { }
105
106             [MethodImpl(MethodImplOptions.AggressiveInlining)]
107             public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
                ↪  transactionId) : this(uniqueTimestampFactory, transactionId, default, default) {
                ↪  }
108
109             [MethodImpl(MethodImplOptions.AggressiveInlining)]
110             public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
                ↪  {After}";
111
112             [MethodImpl(MethodImplOptions.AggressiveInlining)]
113             public override bool Equals(object obj) => obj is Transition transition ?
                ↪  Equals(transition) : false;
114
115             [MethodImpl(MethodImplOptions.AggressiveInlining)]
116             public override int GetHashCode() => (TransactionId, Before, After,
                ↪  Timestamp).GetHashCode();
117
```

```
118             [MethodImpl(MethodImplOptions.AggressiveInlining)]
119             public bool Equals(Transition other) => TransactionId == other.TransactionId &&
                ↪  Before == other.Before && After == other.After && Timestamp == other.Timestamp;
120
121             [MethodImpl(MethodImplOptions.AggressiveInlining)]
122             public static bool operator ==(Transition left, Transition right) =>
                ↪  left.Equals(right);
123
124             [MethodImpl(MethodImplOptions.AggressiveInlining)]
125             public static bool operator !=(Transition left, Transition right) => !(left ==
                ↪  right);
126         }
127
128         /// <remarks>
129         /// Другие варианты реализации транзакций (атомарности):
130         ///     1. Разделение хранения значения связи ((Source Target) или (Source Linker
            ↪  Target)) и индексов.
131         ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
            ↪  потребуется решить вопрос
132         ///        со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
            ↪  пересечениями идентификаторов.
133         ///
134         /// Где хранить промежуточный список транзакций?
135         ///
136         /// В оперативной памяти:
137         ///  Минусы:
138         ///     1. Может усложнить систему, если она будет функционировать самостоятельно,
139         ///     так как нужно отдельно выделять память под список трансформаций.
140         ///     2. Выделенной оперативной памяти может не хватить, в том случае,
141         ///     если транзакция использует слишком много трансформаций.
142         ///         -> Можно использовать жёсткий диск для слишком длинных транзакций.
143         ///         -> Максимальный размер списка трансформаций можно ограничить / задать
            ↪  константой.
144         ///     3. При подтверждении транзакции (Commit) все трансформации записываются разом
            ↪  создавая задержку.
145         ///
146         /// На жёстком диске:
147         ///  Минусы:
148         ///     1. Длительный отклик, на запись каждой трансформации.
149         ///     2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
150         ///         -> Это может решаться упаковкой/исключением дублирующих операций.
151         ///         -> Также это может решаться тем, что короткие транзакции вообще
152         ///            не будут записываться в случае отката.
153         ///     3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
            ↪  операции (трансформации)
154         ///        будут записаны в лог.
155         ///
156         /// </remarks>
157         public class Transaction : DisposableBase
158         {
159             private readonly Queue<Transition> _transitions;
160             private readonly UInt64LinksTransactionsLayer _layer;
161             public bool IsCommitted { get; private set; }
162             public bool IsReverted { get; private set; }
163
164             [MethodImpl(MethodImplOptions.AggressiveInlining)]
165             public Transaction(UInt64LinksTransactionsLayer layer)
166             {
167                 _layer = layer;
168                 if (_layer._currentTransactionId != 0)
169                 {
170                     throw new NotSupportedException("Nested transactions not supported.");
171                 }
172                 IsCommitted = false;
173                 IsReverted = false;
174                 _transitions = new Queue<Transition>();
175                 SetCurrentTransaction(layer, this);
176             }
177
178             [MethodImpl(MethodImplOptions.AggressiveInlining)]
179             public void Commit()
180             {
181                 EnsureTransactionAllowsWriteOperations(this);
182                 while (_transitions.Count > 0)
183                 {
184                     var transition = _transitions.Dequeue();
185                     _layer._transitions.Enqueue(transition);
186                 }
```

```
                _layer._lastCommitedTransactionId = _layer._currentTransactionId;
                IsCommitted = true;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private void Revert()
            {
                EnsureTransactionAllowsWriteOperations(this);
                var transitionsToRevert = new Transition[_transitions.Count];
                _transitions.CopyTo(transitionsToRevert, 0);
                for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
                {
                    _layer.RevertTransition(transitionsToRevert[i]);
                }
                IsReverted = true;
            }

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
            : base(links)
        {
            if (string.IsNullOrWhiteSpace(logAddress))
            {
                throw new ArgumentNullException(nameof(logAddress));
            }
            // В первой строке файла хранится последняя закоммиченную транзакцию.
            // При запуске это используется для проверки удачного закрытия файла лога.
            // In the first line of the file the last committed transaction is stored.
            // On startup, this is used to check that the log file is successfully closed.
            var lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
```

```
            var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
            if (!lastCommitedTransition.Equals(lastWrittenTransition))
            {
                Dispose();
                throw new NotSupportedException("Database is damaged, autorecovery is not
                ↪  supported yet.");
            }
            if (lastCommitedTransition == default)
            {
                FileHelpers.WriteFirst(logAddress, lastCommitedTransition);
            }
            _lastCommitedTransition = lastCommitedTransition;
            // TODO: Think about a better way to calculate or store this value
            var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
            _lastCommitedTransactionId = allTransitions.Length > 0 ? allTransitions.Max(x =>
            ↪  x.TransactionId) : 0;
            _uniqueTimestampFactory = new UniqueTimestampFactory();
            _logAddress = logAddress;
            _log = FileHelpers.Append(logAddress);
            _transitions = new Queue<Transition>();
            _transitionsPusher = new Task(TransitionsPusher);
            _transitionsPusher.Start();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<ulong> GetLinkValue(ulong link) => _links.GetLink(link);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override ulong Create(IList<ulong> restrictions)
        {
            var createdLinkIndex = _links.Create();
            var createdLink = new Link<ulong>(_links.GetLink(createdLinkIndex));
            CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
            ↪  default, createdLink));
            return createdLinkIndex;
        }

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void RevertTransition(Transition transition)
        {
            if (transition.After.IsNull()) // Revert Deletion with Creation
```

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void ResetCurrentTransation()
        {
            _currentTransactionId = 0;
            _currentTransactionTransitions = null;
            _currentTransaction = null;
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void TransitionsPusher()
        {
            while (!Disposable.IsDisposed && _transitionsPusher != null)
            {
                Thread.Sleep(DefaultPushDelay);
                PushTransitions();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Transaction BeginTransaction() => new Transaction(this);

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

        #region DisposalBase

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void Dispose(bool manual, bool wasDisposed)
```

```
415         {
416             if (!wasDisposed)
417             {
418                 DisposeTransitions();
419             }
420             base.Dispose(manual, wasDisposed);
421         }
422 
423         #endregion
424     }
425 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using Platform.Converters;
3  
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5  
6  namespace Platform.Data.Doublets.Unicode
7  {
8      public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
       ↪  IConverter<char, TLink>
9      {
10         private static readonly UncheckedConverter<char, TLink> _charToAddressConverter =
           ↪  UncheckedConverter<char, TLink>.Default;
11 
12         private readonly IConverter<TLink> _addressToNumberConverter;
13         private readonly TLink _unicodeSymbolMarker;
14 
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
           ↪  addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
17         {
18             _addressToNumberConverter = addressToNumberConverter;
19             _unicodeSymbolMarker = unicodeSymbolMarker;
20         }
21 
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public TLink Convert(char source)
24         {
25             var unaryNumber =
               ↪  _addressToNumberConverter.Convert(_charToAddressConverter.Convert(source));
26             return _links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
27         }
28     }
29 }
```

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

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

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

1.109 ./csharp/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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeMap(ILinks<ulong> links) => _links = links;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static UnicodeMap InitNew(ILinks<ulong> links)
        {
            var map = new UnicodeMap(links);
            map.Init();
            return map;
        }

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

```
67
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          public static char FromLinkToChar(ulong link) => (char)(link - 1);
70
71          [MethodImpl(MethodImplOptions.AggressiveInlining)]
72          public static bool IsCharLink(ulong link) => link <= MapSize;
73
74          [MethodImpl(MethodImplOptions.AggressiveInlining)]
75          public static string FromLinksToString(IList<ulong> linksList)
76          {
77              var sb = new StringBuilder();
78              for (int i = 0; i < linksList.Count; i++)
79              {
80                  sb.Append(FromLinkToChar(linksList[i]));
81              }
82              return sb.ToString();
83          }
84
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
87          {
88              var sb = new StringBuilder();
89              if (links.Exists(link))
90              {
91                  StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
92                      x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
                        ↪  element =>
93                      {
94                          sb.Append(FromLinkToChar(element));
95                          return true;
96                      });
97              }
98              return sb.ToString();
99          }
100
101         [MethodImpl(MethodImplOptions.AggressiveInlining)]
102         public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
        ↪  chars.Length);
103
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         public static ulong[] FromCharsToLinkArray(char[] chars, int count)
106         {
107             // char array to ulong array
108             var linksSequence = new ulong[count];
109             for (var i = 0; i < count; i++)
110             {
111                 linksSequence[i] = FromCharToLink(chars[i]);
112             }
113             return linksSequence;
114         }
115
116         [MethodImpl(MethodImplOptions.AggressiveInlining)]
117         public static ulong[] FromStringToLinkArray(string sequence)
118         {
119             // char array to ulong array
120             var linksSequence = new ulong[sequence.Length];
121             for (var i = 0; i < sequence.Length; i++)
122             {
123                 linksSequence[i] = FromCharToLink(sequence[i]);
124             }
125             return linksSequence;
126         }
127
128         [MethodImpl(MethodImplOptions.AggressiveInlining)]
129         public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
130         {
131             var result = new List<ulong[]>();
132             var offset = 0;
133             while (offset < sequence.Length)
134             {
135                 var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
136                 var relativeLength = 1;
137                 var absoluteLength = offset + relativeLength;
138                 while (absoluteLength < sequence.Length &&
139                        currentCategory ==
                           ↪  CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
140                 {
141                     relativeLength++;
142                     absoluteLength++;
```

```
                }
                // char array to ulong array
                var innerSequence = new ulong[relativeLength];
                var maxLength = offset + relativeLength;
                for (var i = offset; i < maxLength; i++)
                {
                    innerSequence[i - offset] = FromCharToLink(sequence[i]);
                }
                result.Add(innerSequence);
                offset += relativeLength;
            }
            return result;
        }

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

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

```
using System;
using System.Linq;
using System.Runtime.CompilerServices;
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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

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

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

```
using System;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Converters;

#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 static readonly UncheckedConverter<TLink, char> _addressToCharConverter =
        ↪  UncheckedConverter<TLink, char>.Default;

        private readonly IConverter<TLink> _numberToAddressConverter;
        private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
        ↪  numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
        ↪  base(links)
        {
            _numberToAddressConverter = numberToAddressConverter;
            _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public char Convert(TLink source)
        {
            if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
            {
                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                ↪  not a unicode symbol.");
            }
            return _addressToCharConverter.Convert(_numberToAddressConverter.Convert(_links.GetS
            ↪  ource(source)));
        }
    }
}
```

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

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

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class GenericLinksTests
    {
```

```
12          [Fact]
13          public static void CRUDTest()
14          {
15              Using<byte>(links => links.TestCRUDOperations());
16              Using<ushort>(links => links.TestCRUDOperations());
17              Using<uint>(links => links.TestCRUDOperations());
18              Using<ulong>(links => links.TestCRUDOperations());
19          }
20
21          [Fact]
22          public static void RawNumbersCRUDTest()
23          {
24              Using<byte>(links => links.TestRawNumbersCRUDOperations());
25              Using<ushort>(links => links.TestRawNumbersCRUDOperations());
26              Using<uint>(links => links.TestRawNumbersCRUDOperations());
27              Using<ulong>(links => links.TestRawNumbersCRUDOperations());
28          }
29
30          [Fact]
31          public static void MultipleRandomCreationsAndDeletionsTest()
32          {
33              Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
                ↪  MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
                ↪  implementation of tree cuts out 5 bits from the address space.
34              Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
                ↪  stMultipleRandomCreationsAndDeletions(100));
35              Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
                ↪  MultipleRandomCreationsAndDeletions(100));
36              Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
                ↪  tMultipleRandomCreationsAndDeletions(100));
37          }
38
39          private static void Using<TLink>(Action<ILinks<TLink>> action)
40          {
41              using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪  UnitedMemoryLinks<TLink>>>())
42              {
43                  action(scope.Use<ILinks<TLink>>());
44              }
45          }
46      }
47  }
```

1.113 ./csharp/Platform.Data.Doublets.Tests/ILinksExtensionsTests.cs

```
1   using Xunit;
2
3   namespace Platform.Data.Doublets.Tests
4   {
5       public class ILinksExtensionsTests
6       {
7           [Fact]
8           public void FormatTest()
9           {
10              using (var scope = new TempLinksTestScope())
11              {
12                  var links = scope.Links;
13                  var link = links.Create();
14                  var linkString = links.Format(link);
15                  Assert.Equal("(1: 1 1)", linkString);
16              }
17          }
18      }
19  }
```

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

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

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

1.115 ./csharp/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.Memory.United.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);

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

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

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

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

```
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 UInt64UnitedMemoryLinks(memory,
                ↪ UInt64UnitedMemoryLinks.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>()
175                 {
176                     UseSequenceMarker = true,
177                     SequenceMarkerLink = unicodeSequenceMarker,
178                     UseIndex = true,
179                     Index = index,
180                     LinksToSequenceConverter = optimalVariantConverter,
181                     Walker = walker,
182                     UseGarbageCollection = true
183                 };
184
185                 var unicodeSequences = new Sequences.Sequences(new
                    ↪ SynchronizedLinks<ulong>(links), unicodeSequencesOptions);
186
187                 // Create some sequences
188                 var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
                    ↪ StringSplitOptions.RemoveEmptyEntries);
189                 var arrays = strings.Select(x => x.Select(y =>
                    ↪ addressToNumberConverter.Convert(y)).ToArray()).ToArray();
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
55
56                  for (var i = 0; i < sequenceLength; i++)
57                  {
58                      links.Delete(sequence[i]);
59                  }
60              }
61          }
62      }
63  }
```

1.117 ./csharp/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.Memory.United.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 UInt64UnitedMemoryLinks(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(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
28              using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
                ↪ UInt64UnitedMemoryLinks.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(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
44              using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
                ↪ UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
45              {
46                  memoryAdapter.TestNonexistentReferences();
47              }
48          }
49
50          private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
51          {
52              var link = memoryAdapter.Create();
53              memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
54              var resultLink = _constants.Null;
55              memoryAdapter.Each(foundLink =>
56              {
57                  resultLink = foundLink[_constants.IndexPart];
58                  return _constants.Break;
59              }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
60              Assert.True(resultLink == link);
61              Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
62              memoryAdapter.Delete(link);
63          }
```

```
    }
}
```

1.118 ./csharp/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.Memory.United.Generic;
using Platform.Data.Doublets.Memory.United.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<UInt64UnitedMemoryLinks>();
                var instance = scope.Use<ILinks<ulong>>();
                Assert.IsType<UInt64UnitedMemoryLinks>(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,
            ↪  UnitedMemoryLinks<ulong>>>())
            {
                var links = scope.Use<ILinks<ulong>>();
                Assert.IsType<UnitedMemoryLinks<ulong>>(links);
            }
        }
    }
}
```

1.119 ./csharp/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);

        //        SequencesOptions o = new SequencesOptions();

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

        //        var sequences = new Sequences(links);

        //        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++)
```

```
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
```

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

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

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

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

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

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

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

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

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

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

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

            var createResults = sequences.CreateAllVariants2(sequence);

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

            var partialSequence = new ulong[sequenceLength - 2];

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

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

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

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

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

            //Global.Trash = searchResults3;

            //var searchResults1Strings = searchResults1.Select(x => x + ": " +
            ↪  sequences.FormatSequence(x)).ToList();
            //Global.Trash = searchResults1Strings;

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

            var intersection2 = createResults.Intersect(searchResults2).ToList();
```

```
            Assert.True(intersection2.Count == createResults.Length);

            var intersection4 = createResults.Intersect(searchResults4).ToList();
            Assert.True(intersection4.Count == createResults.Length);

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

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

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

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

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

            var balancedVariant = balancedVariantConverter.Convert(sequence);

            var partialSequence = new ulong[sequenceLength - 2];

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

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

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

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

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

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

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

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

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

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

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

            var balancedVariant = balancedVariantConverter.Convert(sequence);

            // 1: [1]
            // 2: [2]
```

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

                var doublet = links.GetSource(balancedVariant);

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

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

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

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

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

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

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

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

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

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

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

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

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

                index.Add(sequence);

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

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

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

[![чёрное пространство, белое
↪  пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png
↪  ""чёрное пространство, белое пространство"")](https://raw.githubusercontent.com/Konard/Links
↪  Platform/master/doc/Intro/1.png)

Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая
↪  форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?

[![чёрное пространство, чёрная
↪  точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png
↪  ""чёрное пространство, чёрная
↪  точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png)
```

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

[![белая точка, чёрная точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png ""белая точка, чёрная точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png)

А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой? Гранью? Разделителем? Единицей?

[![две белые точки, чёрная вертикальная линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png ""две белые точки, чёрная вертикальная линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)

Можно ли замкнуть движение? Может ли это быть кругом? Можно ли замкнуть время? Или остаётся только спираль? Но что если замкнуть предел? Создать ограничение, разделение? Получится замкнутая область? Полностью отделённая от всего остального? Но что это всё остальное? Что можно делить? В каком направлении? Ничего или всё? Пустота или полнота? Начало или конец? Или может быть это единица и ноль? Дуальность? Противоположность? А что будет с кругом если у него нет размера? Будет ли круг точкой? Точка состоящая из точек?

[![белая вертикальная линия, чёрный круг](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png ""белая вертикальная линия, чёрный круг"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png)

Как ещё можно использовать грань, черту, линию? А что если она может что-то соединять, может тогда её нужно повернуть? Почему то, что перпендикулярно вертикальному горизонтально? Горизонт? Инвертирует ли это смысл? Что такое смысл? Из чего состоит смысл? Существует ли элементарная единица смысла?

[![белый круг, чёрная горизонтальная линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png ""белый круг, чёрная горизонтальная линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png)

Соединять, допустим, а какой смысл в этом есть ещё? Что если помимо смысла ""соединить, связать"", есть ещё и смысл направления ""от начала к концу""? От предка к потомку? От родителя к ребёнку? От общего к частному?

[![белая горизонтальная линия, чёрная горизонтальная стрелка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png ""белая горизонтальная линия, чёрная горизонтальная стрелка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png)

Шаг назад. Возьмём опять отделённую область, которая лишь та же замкнутая линия, что ещё она может представлять собой? Объект? Но в чём его суть? Разве не в том, что у него есть граница, разделяющая внутреннее и внешнее? Допустим связь, стрелка, линия соединяет два объекта, как бы это выглядело?

[![белая связь, чёрная направленная связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png ""белая связь, чёрная направленная связь"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png)

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

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

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

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

```
422
423 На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
    ↪ рекурсии или фрактала?
424
425 [![белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↪ типизированная связь с двойной рекурсивной внутренней
    ↪ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png
    ↪ ""белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↪ типизированная связь с двойной рекурсивной внутренней структурой"")](https://raw.githubuserc
    ↪ ontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)
426
427 Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
    ↪ Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
428
429 [![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)
430
431 ...
432
433 [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-anima
    ↪ tion-500.gif
    ↪ ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
    ↪ -animation-500.gif)";
434
435         private static readonly string _exampleLoremIpsumText =
436             @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
                ↪ incididunt ut labore et dolore magna aliqua.
437 Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
    ↪ consequat.";
438
439         [Fact]
440         public static void CompressionTest()
441         {
442             using (var scope = new TempLinksTestScope(useSequences: true))
443             {
444                 var links = scope.Links;
445                 var sequences = scope.Sequences;
446
447                 var e1 = links.Create();
448                 var e2 = links.Create();
449
450                 var sequence = new[]
451                 {
452                     e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
453                 };
454
455                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
456                 var totalSequenceSymbolFrequencyCounter = new
                    ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
457                 var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
                    ↪ totalSequenceSymbolFrequencyCounter);
458                 var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
                    ↪ balancedVariantConverter, doubletFrequenciesCache);
459
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
870             //const long sequenceLength = 5; //100% bug
871             const long sequenceLength = 4; //100% bug
872             //const long sequenceLength = 3; //100% _no_bug (ok)
873
874             using (var scope = new TempLinksTestScope(useSequences: true))
875             {
876                 var links = scope.Links;
877                 var sequences = scope.Sequences;
878
879                 var sequence = new ulong[sequenceLength];
880                 for (var i = 0; i < sequenceLength; i++)
881                 {
882                     sequence[i] = links.Create();
883                 }
884
885                 var createResults = sequences.CreateAllVariants2(sequence);
886
887                 Global.Trash = createResults;
```

```
888
889                 for (var i = 0; i < sequenceLength; i++)
890                 {
891                     links.Delete(sequence[i]);
892                 }
893             }
894         }
895
896         [Fact]
897         public static void AllPossibleConnectionsTest()
898         {
899             const long sequenceLength = 5;
900
901             using (var scope = new TempLinksTestScope(useSequences: true))
902             {
903                 var links = scope.Links;
904                 var sequences = scope.Sequences;
905
906                 var sequence = new ulong[sequenceLength];
907                 for (var i = 0; i < sequenceLength; i++)
908                 {
909                     sequence[i] = links.Create();
910                 }
911
912                 var createResults = sequences.CreateAllVariants2(sequence);
913                 var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
914
915                 for (var i = 0; i < 1; i++)
916                 {
917                     var sw1 = Stopwatch.StartNew();
918                     var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
919
920                     var sw2 = Stopwatch.StartNew();
921                     var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
922
923                     var sw3 = Stopwatch.StartNew();
924                     var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
925
926                     var sw4 = Stopwatch.StartNew();
927                     var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
928
929                     Global.Trash = searchResults3;
930                     Global.Trash = searchResults4; //-V3008
931
932                     var intersection1 = createResults.Intersect(searchResults1).ToList();
933                     Assert.True(intersection1.Count == createResults.Length);
934
935                     var intersection2 = reverseResults.Intersect(searchResults1).ToList();
936                     Assert.True(intersection2.Count == reverseResults.Length);
937
938                     var intersection0 = searchResults1.Intersect(searchResults2).ToList();
939                     Assert.True(intersection0.Count == searchResults2.Count);
940
941                     var intersection3 = searchResults2.Intersect(searchResults3).ToList();
942                     Assert.True(intersection3.Count == searchResults3.Count);
943
944                     var intersection4 = searchResults3.Intersect(searchResults4).ToList();
945                     Assert.True(intersection4.Count == searchResults4.Count);
946                 }
947
948                 for (var i = 0; i < sequenceLength; i++)
949                 {
950                     links.Delete(sequence[i]);
951                 }
952             }
953         }
954
955         [Fact(Skip = "Correct implementation is pending")]
956         public static void CalculateAllUsagesTest()
957         {
958             const long sequenceLength = 3;
959
960             using (var scope = new TempLinksTestScope(useSequences: true))
961             {
962                 var links = scope.Links;
963                 var sequences = scope.Sequences;
964
965                 var sequence = new ulong[sequenceLength];
966                 for (var i = 0; i < sequenceLength; i++)
967                 {
```

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

1.120 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs

```
using System;
using Xunit;
using Platform.Memory;
using Platform.Data.Doublets.Memory.Split.Generic;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class SplitMemoryGenericLinksTests
    {
        [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()
        {
            UsingWithExternalReferences<byte>(links => links.TestRawNumbersCRUDOperations());
            UsingWithExternalReferences<ushort>(links => links.TestRawNumbersCRUDOperations());
            UsingWithExternalReferences<uint>(links => links.TestRawNumbersCRUDOperations());
            UsingWithExternalReferences<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 dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory))
            {
```

```
                action(memory);
            }
        }

        private static void UsingWithExternalReferences<TLink>(Action<ILinks<TLink>> action)
        {
            var contants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
            using (var dataMemory = new HeapResizableDirectMemory())
            using (var indexMemory = new HeapResizableDirectMemory())
            using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory,
            ↪  SplitMemoryLinks<TLink>.DefaultLinksSizeStep, contants))
            {
                action(memory);
            }
        }
    }
}
```

1.121 ./csharp/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.Memory.United.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 UInt64UnitedMemoryLinks(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);
        }
    }
}
```

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

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

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;

            var zero = default(T);
            var one = Arithmetic.Increment(zero);

            // Create Link
            Assert.True(equalityComparer.Equals(links.Count(), 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(), one));

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

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

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

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

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

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

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

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

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

            // Delete link
            links.Delete(linkAddress);

            Assert.True(equalityComparer.Equals(links.Count(), zero));

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

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

        public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
        {
            // Constants
            var constants = links.Constants;
            var equalityComparer = EqualityComparer<T>.Default;
```

```
81
82              var zero = default(T);
83              var one = Arithmetic.Increment(zero);
84              var two = Arithmetic.Increment(one);
85
86              var h106E = new Hybrid<T>(106L, isExternal: true);
87              var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
88              var h108E = new Hybrid<T>(-108L);
89
90              Assert.Equal(106L, h106E.AbsoluteValue);
91              Assert.Equal(107L, h107E.AbsoluteValue);
92              Assert.Equal(108L, h108E.AbsoluteValue);
93
94              // Create Link (External -> External)
95              var linkAddress1 = links.Create();
96
97              links.Update(linkAddress1, h106E, h108E);
98
99              var link1 = new Link<T>(links.GetLink(linkAddress1));
100
101             Assert.True(equalityComparer.Equals(link1.Source, h106E));
102             Assert.True(equalityComparer.Equals(link1.Target, h108E));
103
104             // Create Link (Internal -> External)
105             var linkAddress2 = links.Create();
106
107             links.Update(linkAddress2, linkAddress1, h108E);
108
109             var link2 = new Link<T>(links.GetLink(linkAddress2));
110
111             Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
112             Assert.True(equalityComparer.Equals(link2.Target, h108E));
113
114             // Create Link (Internal -> Internal)
115             var linkAddress3 = links.Create();
116
117             links.Update(linkAddress3, linkAddress1, linkAddress2);
118
119             var link3 = new Link<T>(links.GetLink(linkAddress3));
120
121             Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
122             Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
123
124             // Search for created link
125             var setter1 = new Setter<T>(constants.Null);
126             links.Each(h106E, h108E, setter1.SetAndReturnFalse);
127
128             Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
129
130             // Search for nonexistent link
131             var setter2 = new Setter<T>(constants.Null);
132             links.Each(h106E, h107E, setter2.SetAndReturnFalse);
133
134             Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
135
136             // Update link to reference null (prepare for delete)
137             var updated = links.Update(linkAddress3, constants.Null, constants.Null);
138
139             Assert.True(equalityComparer.Equals(updated, linkAddress3));
140
141             link3 = new Link<T>(links.GetLink(linkAddress3));
142
143             Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
144             Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
145
146             // Delete link
147             links.Delete(linkAddress3);
148
149             Assert.True(equalityComparer.Equals(links.Count(), two));
150
151             var setter3 = new Setter<T>(constants.Null);
152             links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
153
154             Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
155         }
156
157         public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
            ↪ links, int maximumOperationsPerCycle)
158         {
159             var comparer = Comparer<TLink>.Default;
```

```
            var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
            var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
            for (var N = 1; N < maximumOperationsPerCycle; N++)
            {
                var random = new System.Random(N);
                var created = 0UL;
                var deleted = 0UL;
                for (var i = 0; i < N; i++)
                {
                    var linksCount = addressToUInt64Converter.Convert(links.Count());
                    var createPoint = random.NextBoolean();
                    if (linksCount > 2 && createPoint)
                    {
                        var linksAddressRange = new Range<ulong>(1, linksCount);
                        TLink source = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA
                        ↪  ddressRange));
                        TLink target = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA
                        ↪  ddressRange));
                        ↪  //-V3086
                        var resultLink = links.GetOrCreate(source, target);
                        if (comparer.Compare(resultLink,
                        ↪  uInt64ToAddressConverter.Convert(linksCount)) > 0)
                        {
                            created++;
                        }
                    }
                    else
                    {
                        links.Create();
                        created++;
                    }
                }
                Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
                for (var i = 0; i < N; i++)
                {
                    TLink link = uInt64ToAddressConverter.Convert((ulong)i + 1UL);
                    if (links.Exists(link))
                    {
                        links.Delete(link);
                        deleted++;
                    }
                }
                Assert.True(addressToUInt64Converter.Convert(links.Count()) == 0L);
            }
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Xunit;
using Platform.Disposables;
using Platform.Ranges;
using Platform.Random;
using Platform.Timestamps;
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.Memory.United.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,
            ↪  UInt64UnitedMemoryLinks>>())
            {
                new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeleti
                ↪  ons(100);
            }
        }

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

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

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

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

                l2 = links.Update(l2, l1);

                links.Delete(l2);

                Global.Trash = links.Count();

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

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

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

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

                links.Delete(l1);

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

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

        [Fact]
        public static void TransactionAutoRevertedTest()
        {
            // Auto Reverted (Because no commit at transaction)
            using (var scope = new TempLinksTestScope(useLog: true))
            {
                var links = scope.Links;
                var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
                using (var transaction = transactionsLayer.BeginTransaction())
                {
                    var l1 = links.Create();
                    var l2 = links.Create();

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

                Assert.Equal(0UL, links.Count());
```

```
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);
```

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

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

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

                    links.Unsync.DisposeIfPossible();

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

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

                        links.Delete(l2);

                        ExceptionThrower();

                        transaction.Commit();
                    }

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

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

                lastScope.DeleteFiles();
            }
        }

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

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

            // Commit
            using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
            using (var links = new UInt64Links(memoryAdapter))
            {
                using (var transaction = memoryAdapter.BeginTransaction())
                {
                    var l1 = links.CreateAndUpdate(itself, itself);
                    var l2 = links.CreateAndUpdate(itself, itself);

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

                    links.Delete(l1);

                    transaction.Commit();
                }

                Global.Trash = links.Count();
            }

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

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

```
253
254             var tempDatabaseFilename = Path.GetTempFileName();
255             var tempTransactionLogFilename = Path.GetTempFileName();
256
257             // Commit
258             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪ UInt64UnitedMemoryLinks(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
                    ↪ UInt64UnitedMemoryLinks(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 UInt64UnitedMemoryLinks(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 UInt64UnitedMemoryLinks(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         {
381             using (var scope = new TempLinksTestScope(useSequences: true))
382             {
383                 var links = scope.Links;
384                 var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
385
386                 var a = links.CreatePoint();
387                 var b = links.CreatePoint();
388                 var c = links.CreatePoint();
389
390                 var ab = links.GetOrCreate(a, b);
391                 var cb = links.GetOrCreate(c, b);
392                 var ac = links.GetOrCreate(a, c);
393
394                 a = links.Update(a, c, b);
395                 b = links.Update(b, a, c);
396                 c = links.Update(c, a, b);
397
398                 Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
399                 Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
400                 Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
```

```
401
402                 Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
                    ↪  "(5:(4:5 (6:5 4)) 6)");
403                 Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
                    ↪  "(6:(5:(4:5 6) 6) 4)");
404                 Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
                    ↪  "(4:(5:4 (6:5 4)) 6)");
405
406                 // 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)"
407
408                 Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
                    ↪  "{{5}{5}{4}{6}}");
409                 Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
                    ↪  "{{5}{6}{6}{4}}");
410                 Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
                    ↪  "{{4}{5}{4}{6}}");
411             }
412         }
413
414         private static void DefaultFormatter(StringBuilder sb, ulong link)
415         {
416             sb.Append(link.ToString());
417         }
418
419         #endregion
420
421         #region Performance
422
423         /*
424        public static void RunAllPerformanceTests()
425        {
426             try
427             {
428                 links.TestLinksInSteps();
429             }
430             catch (Exception ex)
431             {
432                 ex.WriteToConsole();
433             }
434
435             return;
436
437             try
438             {
439                 //ThreadPool.SetMaxThreads(2, 2);
440
441                 // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
    ↪  результат
442                 // Также это дополнительно помогает в отладке
443                 // Увеличивает вероятность попадания информации в кэши
444                 for (var i = 0; i < 10; i++)
445                 {
446                     //0 - 10 ГБ
447                     //Каждые 100 МБ срез цифр
448
449                     //links.TestGetSourceFunction();
450                     //links.TestGetSourceFunctionInParallel();
451                     //links.TestGetTargetFunction();
452                     //links.TestGetTargetFunctionInParallel();
453                     links.Create64BillionLinks();
454
455                     links.TestRandomSearchFixed();
456                     //links.Create64BillionLinksInParallel();
457                     links.TestEachFunction();
458                     //links.TestForeach();
459                     //links.TestParallelForeach();
460                 }
461
462                 links.TestDeletionOfAllLinks();
463
464             }
465             catch (Exception ex)
466             {
467                 ex.WriteToConsole();
468             }
469         }*/
470
471         /*
```

```
472         public static void TestLinksInSteps()
473         {
474             const long gibibyte = 1024 * 1024 * 1024;
475             const long mebibyte = 1024 * 1024;
476 
477             var totalLinksToCreate = gibibyte /
    ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
478             var linksStep = 102 * mebibyte /
    ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
479 
480             var creationMeasurements = new List<TimeSpan>();
481             var searchMeasuremets = new List<TimeSpan>();
482             var deletionMeasurements = new List<TimeSpan>();
483 
484             GetBaseRandomLoopOverhead(linksStep);
485             GetBaseRandomLoopOverhead(linksStep);
486 
487             var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
488 
489             ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
490 
491             var loops = totalLinksToCreate / linksStep;
492 
493             for (int i = 0; i < loops; i++)
494             {
495                 creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
496                 searchMeasuremets.Add(Measure(() => links.RunRandomSearches(linksStep)));
497 
498                 Console.Write("\rC + S {0}/{1}", i + 1, loops);
499             }
500 
501             ConsoleHelpers.Debug();
502 
503             for (int i = 0; i < loops; i++)
504             {
505                 deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
506 
507                 Console.Write("\rD {0}/{1}", i + 1, loops);
508             }
509 
510             ConsoleHelpers.Debug();
511 
512             ConsoleHelpers.Debug("C S D");
513 
514             for (int i = 0; i < loops; i++)
515             {
516                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
    ↪ searchMeasuremets[i], deletionMeasurements[i]);
517             }
518 
519             ConsoleHelpers.Debug("C S D (no overhead)");
520 
521             for (int i = 0; i < loops; i++)
522             {
523                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
    ↪ searchMeasuremets[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
524             }
525 
526             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))
```

```
            {
                long iterations = 64 * 1024 * 1024 /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;

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

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

                var sw = Stopwatch.StartNew();

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

                    counter += links.Search(source, target);
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
    ↪  Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
    ↪  counter);
            }

            File.Delete(tempFilename);
        }*/

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

                var maxLink = links.Count();

                var iterations = links.Count();

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

                var sw = Stopwatch.StartNew();

                for (var i = iterations; i > 0; i--)
                {
                    var linksAddressRange = new
                    ↪  Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);

                    var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
                    var target = RandomHelpers.Default.NextUInt64(linksAddressRange);

                    counter += links.SearchOrDefault(source, target);
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
                ↪  Iterations per second), c: {3}",
                     iterations, elapsedTime, (long)iterationsPerSecond, counter);
            }
        }

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

                var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);

```

```
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 / UInt64UnitedMemoryLinks.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 / UInt64UnitedMemoryLinks.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         }
898
899         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
900         public static void TestDeletionOfAllLinks()
901         {
902             using (var scope = new TempLinksTestScope())
903             {
904                 var links = scope.Links;
905                 var linksBeforeTest = links.Count();
906
907                 ConsoleHelpers.Debug("Deleting all links");
908
909                 var elapsedTime = Performance.Measure(links.DeleteAll);
910
911                 var linksDeleted = linksBeforeTest - links.Count();
912                 var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
913
914                 ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
                    ↪  linksDeleted, elapsedTime,
915                     (long)linksPerSecond);
916             }
917         }
918
```

```
919             #endregion
920         }
921     }
```

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

```
1   using Xunit;
2   using Platform.Random;
3   using Platform.Data.Doublets.Numbers.Unary;
4
5   namespace Platform.Data.Doublets.Tests
6   {
7       public static class UnaryNumberConvertersTests
8       {
9           [Fact]
10          public static void ConvertersTest()
11          {
12              using (var scope = new TempLinksTestScope())
13              {
14                  const int N = 10;
15                  var links = scope.Links;
16                  var meaningRoot = links.CreatePoint();
17                  var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
18                  var powerOf2ToUnaryNumberConverter = new
                    ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
19                  var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
                    ↪  powerOf2ToUnaryNumberConverter);
20                  var random = new System.Random(0);
21                  ulong[] numbers = new ulong[N];
22                  ulong[] unaryNumbers = new ulong[N];
23                  for (int i = 0; i < N; i++)
24                  {
25                      numbers[i] = random.NextUInt64();
26                      unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
27                  }
28                  var fromUnaryNumberConverterUsingOrOperation = new
                    ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                    ↪  powerOf2ToUnaryNumberConverter);
29                  var fromUnaryNumberConverterUsingAddOperation = new
                    ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
30                  for (int i = 0; i < N; i++)
31                  {
32                      Assert.Equal(numbers[i],
                        ↪  fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
33                      Assert.Equal(numbers[i],
                        ↪  fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
34                  }
35              }
36          }
37      }
38  }
```

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

```
1   using Xunit;
2   using Platform.Converters;
3   using Platform.Memory;
4   using Platform.Reflection;
5   using Platform.Scopes;
6   using Platform.Data.Numbers.Raw;
7   using Platform.Data.Doublets.Incrementers;
8   using Platform.Data.Doublets.Numbers.Unary;
9   using Platform.Data.Doublets.PropertyOperators;
10  using Platform.Data.Doublets.Sequences.Converters;
11  using Platform.Data.Doublets.Sequences.Indexes;
12  using Platform.Data.Doublets.Sequences.Walkers;
13  using Platform.Data.Doublets.Unicode;
14  using Platform.Data.Doublets.Memory.United.Generic;
15  using Platform.Data.Doublets.CriterionMatchers;
16
17  namespace Platform.Data.Doublets.Tests
18  {
19      public static class UnicodeConvertersTests
20      {
21          [Fact]
22          public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
23          {
24              using (var scope = new TempLinksTestScope())
25              {
26                  var links = scope.Links;
27                  var meaningRoot = links.CreatePoint();
```

```
28                  var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
29                  var powerOf2ToUnaryNumberConverter = new
                    ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
30                  var addressToUnaryNumberConverter = new
                    ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
31                  var unaryNumberToAddressConverter = new
                    ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                    ↪  powerOf2ToUnaryNumberConverter);
32                  TestCharAndUnicodeSymbolConverters(links, meaningRoot,
                    ↪  addressToUnaryNumberConverter, unaryNumberToAddressConverter);
33              }
34          }
35
36          [Fact]
37          public static void CharAndRawNumberUnicodeSymbolConvertersTest()
38          {
39              using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↪  UnitedMemoryLinks<ulong>>>())
40              {
41                  var links = scope.Use<ILinks<ulong>>();
42                  var meaningRoot = links.CreatePoint();
43                  var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
44                  var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
45                  TestCharAndUnicodeSymbolConverters(links, meaningRoot,
                    ↪  addressToRawNumberConverter, rawNumberToAddressConverter);
46              }
47          }
48
49          private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
            ↪  meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
            ↪  numberToAddressConverter)
50          {
51              var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
52              var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
                ↪  addressToNumberConverter, unicodeSymbolMarker);
53              var originalCharacter = 'H';
54              var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
55              var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
                ↪  unicodeSymbolMarker);
56              var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
                ↪  numberToAddressConverter, unicodeSymbolCriterionMatcher);
57              var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
58              Assert.Equal(originalCharacter, resultingCharacter);
59          }
60
61          [Fact]
62          public static void StringAndUnicodeSequenceConvertersTest()
63          {
64              using (var scope = new TempLinksTestScope())
65              {
66                  var links = scope.Links;
67
68                  var itself = links.Constants.Itself;
69
70                  var meaningRoot = links.CreatePoint();
71                  var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
72                  var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
73                  var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
74                  var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
75                  var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
76
77                  var powerOf2ToUnaryNumberConverter = new
                    ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
78                  var addressToUnaryNumberConverter = new
                    ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
79                  var charToUnicodeSymbolConverter = new
                    ↪  CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
                    ↪  unicodeSymbolMarker);
80
81                  var unaryNumberToAddressConverter = new
                    ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                    ↪  powerOf2ToUnaryNumberConverter);
82                  var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
83                  var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
84                  var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                    ↪  frequencyPropertyMarker, frequencyMarker);
```

```
85                  var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
86                  var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
87                  var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
88                  var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
89
90                  var stringToUnicodeSequenceConverter = new
                    ↪  StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
                    ↪  index, optimalVariantConverter, unicodeSequenceMarker);
91
92                  var originalString = "Hello";
93
94                  var unicodeSequenceLink =
                    ↪  stringToUnicodeSequenceConverter.Convert(originalString);
95
96                  var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
                    ↪  unicodeSymbolMarker);
97                  var unicodeSymbolToCharConverter = new
                    ↪  UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
                    ↪  unicodeSymbolCriterionMatcher);
98
99                  var unicodeSequenceCriterionMatcher = new TargetMatcher<ulong>(links,
                    ↪  unicodeSequenceMarker);
100
101                 var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
                    ↪  unicodeSymbolCriterionMatcher.IsMatched);
102
103                 var unicodeSequenceToStringConverter = new
                    ↪  UnicodeSequenceToStringConverter<ulong>(links,
                    ↪  unicodeSequenceCriterionMatcher, sequenceWalker,
                    ↪  unicodeSymbolToCharConverter);
104
105                 var resultingString =
                    ↪  unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
106
107                 Assert.Equal(originalString, resultingString);
108             }
109         }
110     }
111 }
```

Index