LinksPlatform's Platform.Data.Doublets Class Library

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

```csharp
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

```csharp
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

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

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

namespace Platform.Data.Doublets.Decorators
{
    /// <remarks>
    /// <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);
        }
    }
}
```

```csharp
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);
    }
}
```

```csharp
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
```

```csharp
17              {
18                  [MethodImpl(MethodImplOptions.AggressiveInlining)]
19                  get => true;
20              }
21          }
22
23          protected readonly DisposableWithMultipleCallsAllowed Disposable;
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          protected LinksDisposableDecoratorBase(ILinks<TLink> links) : base(links) => Disposable
    ↪   = new DisposableWithMultipleCallsAllowed(Dispose);
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          ~LinksDisposableDecoratorBase() => Disposable.Destruct();
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          public void Dispose() => Disposable.Dispose();
33
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          protected virtual void Dispose(bool manual, bool wasDisposed)
36          {
37              if (!wasDisposed)
38              {
39                  _links.DisposeIfPossible();
40              }
41          }
42      }
43  }
```

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

```csharp
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Decorators
8   {
9       // TODO: Make LinksExternalReferenceValidator. A layer that checks each link to exist or to
    ↪   be external (hybrid link's raw number).
10      public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
11      {
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
17          {
18              var links = _links;
19              links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
20              return links.Each(handler, restrictions);
21          }
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
25          {
26              // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
27              var links = _links;
28              links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
29              links.EnsureInnerReferenceExists(substitution, nameof(substitution));
30              return links.Update(restrictions, substitution);
31          }
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public override void Delete(IList<TLink> restrictions)
35          {
36              var link = restrictions[_constants.IndexPart];
37              var links = _links;
38              links.EnsureLinkExists(link, nameof(link));
39              links.Delete(link);
40          }
41      }
42  }
```

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

```csharp
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4
```

```csharp
#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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            var constants = _constants;
            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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
14      public override TLink Create(IList<TLink> restrictions) => _links.CreatePoint();

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

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

```csharp
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      public class 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

```csharp
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      public class 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

```csharp
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      public class LinksUsagesValidator<TLink> : LinksDecoratorBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksUsagesValidator(ILinks<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
15         {
16             var links = _links;
17             links.EnsureNoUsages(restrictions[_constants.IndexPart]);
18             return links.Update(restrictions, substitution);
19         }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public override void Delete(IList<TLink> restrictions)
23         {
24             var link = restrictions[_constants.IndexPart];
25             var links = _links;
26             links.EnsureNoUsages(link);
27             links.Delete(link);
28         }
29     }
30 }
```

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

```csharp
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      public class NonNullContentsLinkDeletionResolver<TLink> : LinksDecoratorBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public NonNullContentsLinkDeletionResolver(ILinks<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public override void Delete(IList<TLink> restrictions)
15         {
16             var linkIndex = restrictions[_constants.IndexPart];
17             var links = _links;
18             links.EnforceResetValues(linkIndex);
19             links.Delete(linkIndex);
20         }
21     }
22 }
```

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

```csharp
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      /// <summary>
9      /// <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>
10     /// <para>Представляет комбинированный декоратор, реализующий основную логику по
           взаимодействии с хранилищем связей, для связей с адресами представленными в виде <see
           cref="System.UInt64"/>.</para>
11     /// </summary>
12     /// <remarks>
13     /// Возможные оптимизации:
14     /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
15     ///     + меньше объём БД
16     ///     - меньше производительность
17     ///     - больше ограничение на количество связей в БД)
18     /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
```

```csharp
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

```csharp
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
```

```csharp
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);
```

```
296                    }
297                    return constants.Continue;
298                }
299                else
300                {
301                    throw new NotSupportedException();
302                }
303            }
304        }

305
306        /// <remarks>
307        /// IList[IList[IList[T]]]
308        /// |       |        |      |||
309        /// |       |          ------ ||
310        /// |       |            link  ||
311        /// |        ------------- |
312        /// |             change       |
313        ///   -------------------
314        ///          changes
315        /// </remarks>
316        public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink>
        ↪   substitution)
317        {
318            var changes = new List<IList<IList<TLink>>>();
319            var @continue = _constants.Continue;
320            Trigger(condition, AlwaysContinue, substitution, (before, after) =>
321            {
322                var change = new[] { before, after };
323                changes.Add(change);
324                return @continue;
325            });
326            return changes;
327        }

328
329        private TLink AlwaysContinue(IList<TLink> linkToMatch) => _constants.Continue;
330    }
331 }
```

1.16 ./csharp/Platform.Data.Doublets/Doublet.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets
8  {
9      public struct Doublet<T> : IEquatable<Doublet<T>>
10     {
11         private static readonly EqualityComparer<T> _equalityComparer =
        ↪   EqualityComparer<T>.Default;
12
13         public readonly T Source;
14
15         public readonly T Target;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public Doublet(T source, T target)
19         {
20             Source = source;
21             Target = target;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public override string ToString() => $"{Source}->{Target}";
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
        ↪   && _equalityComparer.Equals(Target, other.Target);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public override bool Equals(object obj) => obj is Doublet<T> doublet ?
        ↪   base.Equals(doublet) : false;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public override int GetHashCode() => (Source, Target).GetHashCode();
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public static bool operator ==(Doublet<T> left, Doublet<T> right) => left.Equals(right);
38
```

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

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

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

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

namespace Platform.Data.Doublets
{
    /// <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

```csharp
#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

```csharp
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;
            for (var i = 0UL; i < amountOfCreations; i++)
            {
                var linksAddressRange = new Range<ulong>(0,
                ↪  addressToUInt64Converter.Convert(links.Count()));
                var source =
                ↪  uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
                var target =
                ↪  uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
                links.GetOrCreate(source, target);
            }
        }
```

```csharp
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             }
```

```csharp
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)]
```

```csharp
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>
```

```csharp
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            }
```

```csharp
        }

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

        /// <summary>
        /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
        ↪ в хранилище связей.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="source">Начало связи.</param>
        /// <param name="target">Конец связи.</param>
        /// <returns>Значение, определяющее существует ли связь.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
        ↪ => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
        ↪ default) > 0;

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
        ↪ restrictions)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            var any = links.Constants.Any;
            for (var i = 0; i < restrictions.Count; i++)
```

```csharp
                {
                    if (!equalityComparer.Equals(restrictions[i], any) &&
                    ↪  !links.Exists(restrictions[i]))
                    {
                        throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                        ↪  $"sequence[{i}]");
                    }
                }
            }

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

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

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

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

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

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

            /// <param name="links">Хранилище связей.</param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
            ↪  params TLink[] addresses)
            {
                var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
                var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
                var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
                ↪  !links.Exists(x)));
                if (nonExistentAddresses.Count > 0)
                {
                    var max = nonExistentAddresses.Max();
```

```csharp
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);
```

```csharp
                    return setter.Result;
            }

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
        ↪  links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            var constants = links.Constants;
            var restrictionsIndex = restrictions[constants.IndexPart];
            var substitutionIndex = substitution[constants.IndexPart];
            if (equalityComparer.Equals(substitutionIndex, default))
            {
                substitutionIndex = restrictionsIndex;
            }
            var source = substitution[constants.SourcePart];
            var target = substitution[constants.TargetPart];
```

```csharp
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
569         /// </summary>
570         /// <param name="links">Хранилище связей.</param>
571         /// <param name="source">Индекс связи, которая является началом на создаваемой
570         ↪   связи.</param>
572         /// <param name="target">Индекс связи, которая является концом для создаваемой
570         ↪   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
575         ↪   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
590         ↪   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
592         ↪   выполняется обновление.</param>
593         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
593         ↪   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,
596         ↪   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,
604                 ↪   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
616         ↪   target)
617         {
618                 var link = links.SearchOrDefault(source, target);
619                 if (!EqualityComparer<TLink>.Default.Equals(link, default))
620                 {
621                         links.Delete(link);
622                         return link;
623                 }
624                 return default;
625         }
626
627         /// <summary>Удаляет несколько связей.</summary>
628         /// <param name="links">Хранилище связей.</param>
```

```csharp
        /// <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)
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
        {
            if (!links.AreValuesReset(linkIndex))
            {
                links.ResetValues(linkIndex);
            }
        }
```

```
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);
```

```csharp
                    links.Delete(oldLinkIndex);
                }
                return newLinkIndex;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static ILinks<TLink>
            ↪  DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
            {
                links = new LinksCascadeUsagesResolver<TLink>(links);
                links = new NonNullContentsLinkDeletionResolver<TLink>(links);
                links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
                return links;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static string Format<TLink>(this ILinks<TLink> links, IList<TLink> link)
            {
                var constants = links.Constants;
                return $"({link[constants.IndexPart]}: {link[constants.SourcePart]}
                ↪  {link[constants.TargetPart]})";
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static string Format<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪  links.Format(links.GetLink(link));
        }
    }
```

1.20 ./csharp/Platform.Data.Doublets/ISynchronizedLinks.cs

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

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

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

```csharp
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 FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Increment(TLink frequency)
        {
            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);
```

```
36              }
37          }
38  }
```

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

```csharp
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 UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
10      {
11          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
12
13          private readonly TLink _unaryOne;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
            ↪  _unaryOne = unaryOne;
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          public TLink Increment(TLink unaryNumber)
20          {
21              var links = _links;
22              if (_equalityComparer.Equals(unaryNumber, _unaryOne))
23              {
24                  return links.GetOrCreate(_unaryOne, _unaryOne);
25              }
26              var source = links.GetSource(unaryNumber);
27              var target = links.GetTarget(unaryNumber);
28              if (_equalityComparer.Equals(source, target))
29              {
30                  return links.GetOrCreate(unaryNumber, _unaryOne);
31              }
32              else
33              {
34                  return links.GetOrCreate(source, Increment(target));
35              }
36          }
37      }
38  }
```

1.23 ./csharp/Platform.Data.Doublets/Link.cs

```csharp
1   using Platform.Collections.Lists;
2   using Platform.Exceptions;
3   using Platform.Ranges;
4   using Platform.Singletons;
5   using System;
6   using System.Collections;
7   using System.Collections.Generic;
8   using System.Runtime.CompilerServices;
9
10  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12  namespace Platform.Data.Doublets
13  {
14      /// <summary>
15      /// Структура описывающая уникальную связь.
16      /// </summary>
17      public struct Link<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>, IList<TLink>
18      {
19          public static readonly Link<TLink> Null = new Link<TLink>();
20
21          private static readonly LinksConstants<TLink> _constants =
            ↪  Default<LinksConstants<TLink>>.Instance;
22          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
23
24          private const int Length = 3;
25
26          public readonly TLink Index;
27          public readonly TLink Source;
28          public readonly TLink Target;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public Link(params TLink[] values) => SetValues(values, out Index, out Source, out
            ↪  Target);
```

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

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);

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

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

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

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

128          [MethodImpl(MethodImplOptions.AggressiveInlining)]
129          public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
       ↪   ToString(Source, Target) : ToString(Index, Source, Target);

131          #region IList

133          public int Count
134          {
135              [MethodImpl(MethodImplOptions.AggressiveInlining)]
136              get => Length;
137          }

139          public bool IsReadOnly
140          {
141              [MethodImpl(MethodImplOptions.AggressiveInlining)]
142              get => true;
143          }

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          }

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

172          [MethodImpl(MethodImplOptions.AggressiveInlining)]
173          public IEnumerator<TLink> GetEnumerator()
174          {
175              yield return Index;
176              yield return Source;
177              yield return Target;
178          }

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)]
10         public static bool IsFullPoint<TLink>(this Link<TLink> link) =>
              ↪  Point<TLink>.IsFullPoint(link);
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
              ↪  Point<TLink>.IsPartialPoint(link);
14     }
15 }
```

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

```csharp
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

```csharp
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

```csharp
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/IndexTreeType.cs

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

namespace Platform.Data.Doublets.Memory
{
    public enum IndexTreeType
    {
        Default = 0,
        SizeBalancedTree = 1,
        RecursionlessSizeBalancedTree = 2,
        SizedAndThreadedAVLBalancedTree = 3
    }
}
```

```csharp
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Unsafe;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Memory
9   {
10      public struct LinksHeader<TLink> : IEquatable<LinksHeader<TLink>>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
              ↪  EqualityComparer<TLink>.Default;
13
14          public static readonly long SizeInBytes = Structure<LinksHeader<TLink>>.Size;
15
16          public TLink AllocatedLinks;
17          public TLink ReservedLinks;
18          public TLink FreeLinks;
19          public TLink FirstFreeLink;
20          public TLink RootAsSource;
21          public TLink RootAsTarget;
22          public TLink LastFreeLink;
23          public TLink Reserved8;
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          public override bool Equals(object obj) => obj is LinksHeader<TLink> linksHeader ?
              ↪  Equals(linksHeader) : false;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          public bool Equals(LinksHeader<TLink> other)
30              => _equalityComparer.Equals(AllocatedLinks, other.AllocatedLinks)
31              && _equalityComparer.Equals(ReservedLinks, other.ReservedLinks)
32              && _equalityComparer.Equals(FreeLinks, other.FreeLinks)
33              && _equalityComparer.Equals(FirstFreeLink, other.FirstFreeLink)
34              && _equalityComparer.Equals(RootAsSource, other.RootAsSource)
35              && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
36              && _equalityComparer.Equals(LastFreeLink, other.LastFreeLink)
37              && _equalityComparer.Equals(Reserved8, other.Reserved8);
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          public override int GetHashCode() => (AllocatedLinks, ReservedLinks, FreeLinks,
              ↪  FirstFreeLink, RootAsSource, RootAsTarget, LastFreeLink, Reserved8).GetHashCode();
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          public static bool operator ==(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
              ↪  left.Equals(right);
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          public static bool operator !=(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
              ↪  !(left == right);
47      }
48  }
```

```csharp
1   using System;
2   using System.Text;
3   using System.Collections.Generic;
4   using System.Runtime.CompilerServices;
5   using Platform.Collections.Methods.Trees;
6   using Platform.Converters;
7   using static System.Runtime.CompilerServices.Unsafe;
8
9   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11  namespace Platform.Data.Doublets.Memory.Split.Generic
12  {
13      public unsafe abstract class ExternalLinksSizeBalancedTreeMethodsBase<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)]
```

```csharp
24          protected ExternalLinksSizeBalancedTreeMethodsBase(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();
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected abstract TLink GetBasePartValue(TLink link);
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
        ↪   rootSource, TLink rootTarget);
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
        ↪   rootSource, TLink rootTarget);
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
        ↪   AsRef<LinksHeader<TLink>>(Header);
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
        ↪   AsRef<RawLinkDataPart<TLink>>(LinksDataParts + RawLinkDataPart<TLink>.SizeInBytes *
        ↪   _addressToInt64Converter.Convert(link));
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
        ↪   ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
        ↪   RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link));
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          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
63          {
64              ref var firstLink = ref GetLinkDataPartReference(first);
65              ref var secondLink = ref GetLinkDataPartReference(second);
66              return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
        ↪   secondLink.Source, secondLink.Target);
67          }
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
71          {
72              ref var firstLink = ref GetLinkDataPartReference(first);
73              ref var secondLink = ref GetLinkDataPartReference(second);
74              return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
        ↪   secondLink.Source, secondLink.Target);
75          }
76
77          public TLink this[TLink index]
78          {
79              [MethodImpl(MethodImplOptions.AggressiveInlining)]
80              get
81              {
82                  var root = GetTreeRoot();
83                  if (GreaterOrEqualThan(index, GetSize(root)))
84                  {
85                      return Zero;
86                  }
87                  while (!EqualToZero(root))
88                  {
89                      var left = GetLeftOrDefault(root);
90                      var leftSize = GetSizeOrZero(left);
91                      if (LessThan(index, leftSize))
```

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

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

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

```csharp
using System.Runtime.CompilerServices;

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

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

```csharp
      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;
          }
      }
  }
```

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

```csharp
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>
    {
```

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

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

```csharp
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
{
```

```csharp
public unsafe abstract class InternalLinksSizeBalancedTreeMethodsBase<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 InternalLinksSizeBalancedTreeMethodsBase(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(TLink link);

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

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

    [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 override bool FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
        LessThan(GetKeyPartValue(first), GetKeyPartValue(second));

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

    [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);
    }

    public TLink this[TLink link, TLink index]
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        get
        {
            var root = GetTreeRoot(link);
            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;
```

```csharp
83                      }
84                      root = GetRightOrDefault(root);
85                      index = Subtract(index, Increment(leftSize));
86                  }
87                  return Zero; // TODO: Impossible situation exception (only if tree structure
   ↪   broken)
88              }
89          }

90
91          /// <summary>
92          /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
   ↪   (концом).
93          /// </summary>
94          /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
95          /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
96          /// <returns>Индекс искомой связи.</returns>
97          [MethodImpl(MethodImplOptions.AggressiveInlining)]
98          public abstract TLink Search(TLink source, TLink target);

99
100         [MethodImpl(MethodImplOptions.AggressiveInlining)]
101         protected TLink SearchCore(TLink root, TLink key)
102         {
103             while (!EqualToZero(root))
104             {
105                 var rootKey = GetKeyPartValue(root);
106                 if (LessThan(key, rootKey)) // node.Key < root.Key
107                 {
108                     root = GetLeftOrDefault(root);
109                 }
110                 else if (GreaterThan(key, rootKey)) // node.Key > root.Key
111                 {
112                     root = GetRightOrDefault(root);
113                 }
114                 else // node.Key == root.Key
115                 {
116                     return root;
117                 }
118             }
119             return Zero;
120         }

121
122         // TODO: Return indices range instead of references count
123         [MethodImpl(MethodImplOptions.AggressiveInlining)]
124         public TLink CountUsages(TLink link) => GetSizeOrZero(GetTreeRoot(link));

125
126         [MethodImpl(MethodImplOptions.AggressiveInlining)]
127         public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
   ↪   EachUsageCore(@base, GetTreeRoot(@base), handler);

128
129         // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
   ↪   low-level MSIL stack.
130         [MethodImpl(MethodImplOptions.AggressiveInlining)]
131         private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
132         {
133             var @continue = Continue;
134             if (EqualToZero(link))
135             {
136                 return @continue;
137             }
138             var @break = Break;
139             if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
140             {
141                 return @break;
142             }
143             if (AreEqual(handler(GetLinkValues(link)), @break))
144             {
145                 return @break;
146             }
147             if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
148             {
149                 return @break;
150             }
151             return @continue;
152         }

153
154         [MethodImpl(MethodImplOptions.AggressiveInlining)]
155         protected override void PrintNodeValue(TLink node, StringBuilder sb)
156         {
157             ref var link = ref GetLinkDataPartReference(node);
```

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

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

```csharp
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);
```

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

```csharp
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.Split.Generic
6  {
7      public unsafe class InternalLinksTargetsSizeBalancedTreeMethods<TLink> :
         ↪ InternalLinksSizeBalancedTreeMethodsBase<TLink>
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public InternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants,
            ↪ byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
            ↪ linksDataParts, linksIndexParts, header) { }
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         protected override ref TLink GetLeftReference(TLink node) => ref
            ↪ GetLinkIndexPartReference(node).LeftAsTarget;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected override ref TLink GetRightReference(TLink node) => ref
            ↪ GetLinkIndexPartReference(node).RightAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override TLink GetLeft(TLink node) =>
            ↪ GetLinkIndexPartReference(node).LeftAsTarget;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override TLink GetRight(TLink node) =>
            ↪ GetLinkIndexPartReference(node).RightAsTarget;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override void SetLeft(TLink node, TLink left) =>
            ↪ GetLinkIndexPartReference(node).LeftAsTarget = left;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override void SetRight(TLink node, TLink right) =>
            ↪ GetLinkIndexPartReference(node).RightAsTarget = right;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override TLink GetSize(TLink node) =>
            ↪ GetLinkIndexPartReference(node).SizeAsTarget;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override void SetSize(TLink node, TLink size) =>
            ↪ GetLinkIndexPartReference(node).SizeAsTarget = size;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override TLink GetTreeRoot(TLink link) =>
            ↪ GetLinkIndexPartReference(link).RootAsTarget;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override TLink GetBasePartValue(TLink link) =>
            ↪ GetLinkDataPartReference(link).Target;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override TLink GetKeyPartValue(TLink link) =>
            ↪ GetLinkDataPartReference(link).Source;
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override void ClearNode(TLink node)
47         {
48             ref var link = ref GetLinkIndexPartReference(node);
49             link.LeftAsTarget = Zero;
50             link.RightAsTarget = Zero;
51             link.SizeAsTarget = Zero;
52         }
53
54         public override TLink Search(TLink source, TLink target) =>
            ↪ SearchCore(GetTreeRoot(target), source);
55     }
56 }
```

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

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

```csharp
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));
```

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

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

```csharp
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Disposables;
using Platform.Singletons;
using Platform.Converters;
using Platform.Numbers;
using Platform.Memory;
using Platform.Data.Exceptions;

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

namespace Platform.Data.Doublets.Memory.Split.Generic
{
    public abstract class SplitMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪   EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
        ↪   UncheckedConverter<TLink, long>.Default;
        private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
        ↪   UncheckedConverter<long, TLink>.Default;

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

        /// <summary>Возвращает размер одной связи в байтах.</summary>
        /// <remarks>
        /// Используется только во вне класса, не рекомедуется использовать внутри.
        /// Так как во вне не обязательно будет доступен unsafe C#.
        /// </remarks>
        public static readonly long LinkDataPartSizeInBytes = RawLinkDataPart<TLink>.SizeInBytes;

        public static readonly long LinkIndexPartSizeInBytes =
        ↪   RawLinkIndexPart<TLink>.SizeInBytes;

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

        public static readonly long DefaultLinksSizeStep = 1 * 1024 * 1024;

        protected readonly IResizableDirectMemory _dataMemory;
        protected readonly IResizableDirectMemory _indexMemory;
        protected readonly long _dataMemoryReservationStepInBytes;
        protected readonly long _indexMemoryReservationStepInBytes;

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

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

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

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪  indexMemory, long memoryReservationStep, LinksConstants<TLink> constants)
        {
            _dataMemory = dataMemory;
            _indexMemory = indexMemory;
            _dataMemoryReservationStepInBytes = memoryReservationStep * LinkDataPartSizeInBytes;
            _indexMemoryReservationStepInBytes = memoryReservationStep *
            ↪  LinkIndexPartSizeInBytes;
            Constants = constants;
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void Init(IResizableDirectMemory dataMemory, IResizableDirectMemory
        ↪  indexMemory, long memoryReservationStep)
        {
            if (dataMemory.ReservedCapacity < memoryReservationStep)
            {
                dataMemory.ReservedCapacity = memoryReservationStep;
            }
            if (indexMemory.ReservedCapacity < memoryReservationStep)
            {
                indexMemory.ReservedCapacity = memoryReservationStep;
            }
            SetPointers(dataMemory, indexMemory);
            ref var header = ref GetHeaderReference();
            // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
            // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
            dataMemory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) *
            ↪  LinkDataPartSizeInBytes + LinkDataPartSizeInBytes; // First link is read only
            ↪  zero link.
            indexMemory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) *
            ↪  LinkIndexPartSizeInBytes + LinkHeaderSizeInBytes;
            // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
            // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
            header.ReservedLinks = ConvertToAddress((dataMemory.ReservedCapacity -
            ↪  LinkDataPartSizeInBytes) / LinkDataPartSizeInBytes);
        }

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

```csharp
                                    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
```

```
358                                  {
359                                      return InternalTargetsTreeMethods.EachUsage(target, handler);
360                                  }
361                              }
362                          else if (AreEqual(target, any))
363                          {
364                              if (externalReferencesRange.HasValue &&
                                 ↪  externalReferencesRange.Value.Contains(source))
365                              {
366                                  return ExternalSourcesTreeMethods.EachUsage(source, handler);
367                              }
368                              else
369                              {
370                                  return InternalSourcesTreeMethods.EachUsage(source, handler);
371                              }
372                          }
373                          else //if(source != Any && target != Any)
374                          {
375                              TLink link;
376                              if (externalReferencesRange.HasValue)
377                              {
378                                  if (externalReferencesRange.Value.Contains(source) &&
                                     ↪  externalReferencesRange.Value.Contains(target))
379                                  {
380                                      link = ExternalSourcesTreeMethods.Search(source, target);
381                                  }
382                                  else if (externalReferencesRange.Value.Contains(source))
383                                  {
384                                      link = InternalTargetsTreeMethods.Search(source, target);
385                                  }
386                                  else if (externalReferencesRange.Value.Contains(target))
387                                  {
388                                      link = InternalSourcesTreeMethods.Search(source, target);
389                                  }
390                                  else
391                                  {
392                                      if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                                         ↪  InternalTargetsTreeMethods.CountUsages(target)))
393                                      {
394                                          link = InternalTargetsTreeMethods.Search(source, target);
395                                      }
396                                      else
397                                      {
398                                          link = InternalSourcesTreeMethods.Search(source, target);
399                                      }
400                                  }
401                              }
402                              else
403                              {
404                                  if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
                                     ↪  InternalTargetsTreeMethods.CountUsages(target)))
405                                  {
406                                      link = InternalTargetsTreeMethods.Search(source, target);
407                                  }
408                                  else
409                                  {
410                                      link = InternalSourcesTreeMethods.Search(source, target);
411                                  }
412                              }
413                              return AreEqual(link, constants.Null) ? @continue :
                                 ↪  handler(GetLinkStruct(link));
414                          }
415                      }
416                  else
417                  {
418                      if (!Exists(index))
419                      {
420                          return @continue;
421                      }
422                      if (AreEqual(source, any) && AreEqual(target, any))
423                      {
424                          return handler(GetLinkStruct(index));
425                      }
426                      ref var storedLinkValue = ref GetLinkDataPartReference(index);
427                      if (!AreEqual(source, any) && !AreEqual(target, any))
428                      {
429                          if (AreEqual(storedLinkValue.Source, source) &&
430                              AreEqual(storedLinkValue.Target, target))
```

```
431                          {
432                              return handler(GetLinkStruct(index));
433                          }
434                          return @continue;
435                      }
436                      var value = default(TLink);
437                      if (AreEqual(source, any))
438                      {
439                          value = target;
440                      }
441                      if (AreEqual(target, any))
442                      {
443                          value = source;
444                      }
445                      if (AreEqual(storedLinkValue.Source, value) ||
446                          AreEqual(storedLinkValue.Target, value))
447                      {
448                          return handler(GetLinkStruct(index));
449                      }
450                      return @continue;
451                  }
452              }
453              throw new NotSupportedException("Другие размеры и способы ограничений не
       ↪    поддерживаются.");
454          }
455
456          /// <remarks>
457          /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
       ↪    в другом месте (но не в менеджере памяти, а в логике Links)
458          /// </remarks>
459          [MethodImpl(MethodImplOptions.AggressiveInlining)]
460          public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
461          {
462              var constants = Constants;
463              var @null = constants.Null;
464              var externalReferencesRange = constants.ExternalReferencesRange;
465              var linkIndex = restrictions[constants.IndexPart];
466              ref var link = ref GetLinkDataPartReference(linkIndex);
467              var source = link.Source;
468              var target = link.Target;
469              ref var header = ref GetHeaderReference();
470              ref var rootAsSource = ref header.RootAsSource;
471              ref var rootAsTarget = ref header.RootAsTarget;
472              // Будет корректно работать только в том случае, если пространство выделенной связи
       ↪    предварительно заполнено нулями
473              if (!AreEqual(source, @null))
474              {
475                  if (externalReferencesRange.HasValue &&
       ↪    externalReferencesRange.Value.Contains(source))
476                  {
477                      ExternalSourcesTreeMethods.Detach(ref rootAsSource, linkIndex);
478                  }
479                  else
480                  {
481                      InternalSourcesTreeMethods.Detach(ref
       ↪    GetLinkIndexPartReference(source).RootAsSource, linkIndex);
482                  }
483              }
484              if (!AreEqual(target, @null))
485              {
486                  if (externalReferencesRange.HasValue &&
       ↪    externalReferencesRange.Value.Contains(target))
487                  {
488                      ExternalTargetsTreeMethods.Detach(ref rootAsTarget, linkIndex);
489                  }
490                  else
491                  {
492                      InternalTargetsTreeMethods.Detach(ref
       ↪    GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
493                  }
494              }
495              source = link.Source = substitution[constants.SourcePart];
496              target = link.Target = substitution[constants.TargetPart];
497              if (!AreEqual(source, @null))
498              {
499                  if (externalReferencesRange.HasValue &&
       ↪    externalReferencesRange.Value.Contains(source))
500                  {
```

```csharp
                    ExternalSourcesTreeMethods.Attach(ref rootAsSource, linkIndex);
                }
                else
                {
                    InternalSourcesTreeMethods.Attach(ref
                    ↪ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
                }
            }
            if (!AreEqual(target, @null))
            {
                if (externalReferencesRange.HasValue &&
                ↪ externalReferencesRange.Value.Contains(target))
                {
                    ExternalTargetsTreeMethods.Attach(ref rootAsTarget, linkIndex);
                }
                else
                {
                    InternalTargetsTreeMethods.Attach(ref
                    ↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
                }
            }
            return linkIndex;
        }

        /// <remarks>
        /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
        ↪ пространство
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Create(IList<TLink> restrictions)
        {
            ref var header = ref GetHeaderReference();
            var freeLink = header.FirstFreeLink;
            if (!AreEqual(freeLink, Constants.Null))
            {
                UnusedLinksListMethods.Detach(freeLink);
            }
            else
            {
                var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
                if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
                {
                    throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
                }
                if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
                {
                    _dataMemory.ReservedCapacity += _dataMemoryReservationStepInBytes;
                    _indexMemory.ReservedCapacity += _indexMemoryReservationStepInBytes;
                    SetPointers(_dataMemory, _indexMemory);
                    header = ref GetHeaderReference();
                    header.ReservedLinks = ConvertToAddress(_dataMemory.ReservedCapacity /
                    ↪ LinkDataPartSizeInBytes);
                }
                header.AllocatedLinks = Increment(header.AllocatedLinks);
                _dataMemory.UsedCapacity += LinkDataPartSizeInBytes;
                _indexMemory.UsedCapacity += LinkIndexPartSizeInBytes;
                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);
                _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
                _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
                // 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)
```

```
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)]
```

```csharp
        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();
                _dataMemory.DisposeIfPossible();
                _indexMemory.DisposeIfPossible();
            }
        }

        #endregion
    }
}
```

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

```csharp
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.Split.Generic
{
    public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
    ↪  ILinksListMethods<TLink>
```

```csharp
    {
        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;
    }
}
```

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

```csharp
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;
```

```
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          public override bool Equals(object obj) => obj is RawLinkDataPart<TLink> link ?
   ↪  Equals(link) : false;
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          public bool Equals(RawLinkDataPart<TLink> other)
24              => _equalityComparer.Equals(Source, other.Source)
25              && _equalityComparer.Equals(Target, other.Target);
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          public override int GetHashCode() => (Source, Target).GetHashCode();
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public static bool operator ==(RawLinkDataPart<TLink> left, RawLinkDataPart<TLink>
   ↪  right) => left.Equals(right);
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public static bool operator !=(RawLinkDataPart<TLink> left, RawLinkDataPart<TLink>
   ↪  right) => !(left == right);
35      }
36  }
```

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

```
1  using Platform.Unsafe;
2  using System;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Memory.Split
9  {
10      public struct RawLinkIndexPart<TLink> : IEquatable<RawLinkIndexPart<TLink>>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
   ↪  EqualityComparer<TLink>.Default;
13
14          public static readonly long SizeInBytes = Structure<RawLinkIndexPart<TLink>>.Size;
15
16          public TLink RootAsSource;
17          public TLink LeftAsSource;
18          public TLink RightAsSource;
19          public TLink SizeAsSource;
20          public TLink RootAsTarget;
21          public TLink LeftAsTarget;
22          public TLink RightAsTarget;
23          public TLink SizeAsTarget;
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          public override bool Equals(object obj) => obj is RawLinkIndexPart<TLink> link ?
   ↪  Equals(link) : false;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          public bool Equals(RawLinkIndexPart<TLink> other)
30              => _equalityComparer.Equals(RootAsSource, other.RootAsSource)
31              && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
32              && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
33              && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
34              && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
35              && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
36              && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
37              && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          public override int GetHashCode() => (RootAsSource, LeftAsSource, RightAsSource,
   ↪  SizeAsSource, RootAsTarget, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          public static bool operator ==(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
   ↪  right) => left.Equals(right);
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          public static bool operator !=(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
   ↪  right) => !(left == right);
47      }
48  }
```

```csharp
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);
```

```csharp
67            }
68
69            [MethodImpl(MethodImplOptions.AggressiveInlining)]
70            protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
71            {
72                ref var firstLink = ref GetLinkReference(first);
73                ref var secondLink = ref GetLinkReference(second);
74                return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                   ↪  secondLink.Source, secondLink.Target);
75            }
76
77            [MethodImpl(MethodImplOptions.AggressiveInlining)]
78            protected virtual TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5,
                ↪  -5);
79
80            [MethodImpl(MethodImplOptions.AggressiveInlining)]
81            protected virtual void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
                ↪  Bit<TLink>.PartialWrite(storedValue, size, 5, -5);
82
83            [MethodImpl(MethodImplOptions.AggressiveInlining)]
84            protected virtual bool GetLeftIsChildValue(TLink value)
85            {
86                unchecked
87                {
88                    return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 4, 1));
89                    //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
90                }
91            }
92
93            [MethodImpl(MethodImplOptions.AggressiveInlining)]
94            protected virtual void SetLeftIsChildValue(ref TLink storedValue, bool value)
95            {
96                unchecked
97                {
98                    var previousValue = storedValue;
99                    var modified = Bit<TLink>.PartialWrite(previousValue,
                       ↪  _boolToAddressConverter.Convert(value), 4, 1);
100                   storedValue = modified;
101               }
102           }
103
104           [MethodImpl(MethodImplOptions.AggressiveInlining)]
105           protected virtual bool GetRightIsChildValue(TLink value)
106           {
107               unchecked
108               {
109                   return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 3, 1));
110                   //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
111               }
112           }
113
114           [MethodImpl(MethodImplOptions.AggressiveInlining)]
115           protected virtual void SetRightIsChildValue(ref TLink storedValue, bool value)
116           {
117               unchecked
118               {
119                   var previousValue = storedValue;
120                   var modified = Bit<TLink>.PartialWrite(previousValue,
                      ↪  _boolToAddressConverter.Convert(value), 3, 1);
121                   storedValue = modified;
122               }
123           }
124
125           [MethodImpl(MethodImplOptions.AggressiveInlining)]
126           protected bool IsChild(TLink parent, TLink possibleChild)
127           {
128               var parentSize = GetSize(parent);
129               var childSize = GetSizeOrZero(possibleChild);
130               return GreaterThanZero(childSize) && LessOrEqualThan(childSize, parentSize);
131           }
132
133           [MethodImpl(MethodImplOptions.AggressiveInlining)]
134           protected virtual sbyte GetBalanceValue(TLink storedValue)
135           {
136               unchecked
137               {
138                   var value = _addressToInt32Converter.Convert(Bit<TLink>.PartialRead(storedValue,
                      ↪  0, 3));
```

```
139              value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
      ↪  end of sbyte
140              return (sbyte)value;
141          }
142      }
143
144      [MethodImpl(MethodImplOptions.AggressiveInlining)]
145      protected virtual void SetBalanceValue(ref TLink storedValue, sbyte value)
146      {
147          unchecked
148          {
149              var packagedValue = _int32ToAddressConverter.Convert((byte)value >> 5 & 4 |
      ↪  value & 3);
150              var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
151              storedValue = modified;
152          }
153      }
154
155      public TLink this[TLink index]
156      {
157          [MethodImpl(MethodImplOptions.AggressiveInlining)]
158          get
159          {
160              var root = GetTreeRoot();
161              if (GreaterOrEqualThan(index, GetSize(root)))
162              {
163                  return Zero;
164              }
165              while (!EqualToZero(root))
166              {
167                  var left = GetLeftOrDefault(root);
168                  var leftSize = GetSizeOrZero(left);
169                  if (LessThan(index, leftSize))
170                  {
171                      root = left;
172                      continue;
173                  }
174                  if (AreEqual(index, leftSize))
175                  {
176                      return root;
177                  }
178                  root = GetRightOrDefault(root);
179                  index = Subtract(index, Increment(leftSize));
180              }
181              return Zero; // TODO: Impossible situation exception (only if tree structure
      ↪  broken)
182          }
183      }
184
185      /// <summary>
186      /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
      ↪  (концом).
187      /// </summary>
188      /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
189      /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
190      /// <returns>Индекс искомой связи.</returns>
191      [MethodImpl(MethodImplOptions.AggressiveInlining)]
192      public TLink Search(TLink source, TLink target)
193      {
194          var root = GetTreeRoot();
195          while (!EqualToZero(root))
196          {
197              ref var rootLink = ref GetLinkReference(root);
198              var rootSource = rootLink.Source;
199              var rootTarget = rootLink.Target;
200              if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
      ↪  node.Key < root.Key
201              {
202                  root = GetLeftOrDefault(root);
203              }
204              else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
      ↪  node.Key > root.Key
205              {
206                  root = GetRightOrDefault(root);
207              }
208              else // node.Key == root.Key
209              {
210                  return root;
```

```csharp
                }
            }
            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);
```

```
290                        if (EqualToZero(current) || !AreEqual(GetBasePartValue(current), link))
291                        {
292                            break;
293                        }
294                    }
295                }
296                return Continue;
297            }
298
299            [MethodImpl(MethodImplOptions.AggressiveInlining)]
300            protected override void PrintNodeValue(TLink node, StringBuilder sb)
301            {
302                ref var link = ref GetLinkReference(node);
303                sb.Append(' ');
304                sb.Append(link.Source);
305                sb.Append('-');
306                sb.Append('>');
307                sb.Append(link.Target);
308            }
309        }
310    }
```

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

```csharp
1    using System;
2    using System.Text;
3    using System.Collections.Generic;
4    using System.Runtime.CompilerServices;
5    using Platform.Collections.Methods.Trees;
6    using Platform.Converters;
7    using static System.Runtime.CompilerServices.Unsafe;
8
9    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11   namespace Platform.Data.Doublets.Memory.United.Generic
12   {
13       public unsafe abstract class LinksSizeBalancedTreeMethodsBase<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* Links;
20           protected readonly byte* Header;
21
22           [MethodImpl(MethodImplOptions.AggressiveInlining)]
23           protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
             ↪   byte* header)
24           {
25               Links = links;
26               Header = header;
27               Break = constants.Break;
28               Continue = constants.Continue;
29           }
30
31           [MethodImpl(MethodImplOptions.AggressiveInlining)]
32           protected abstract TLink GetTreeRoot();
33
34           [MethodImpl(MethodImplOptions.AggressiveInlining)]
35           protected abstract TLink GetBasePartValue(TLink link);
36
37           [MethodImpl(MethodImplOptions.AggressiveInlining)]
38           protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
             ↪   rootSource, TLink rootTarget);
39
40           [MethodImpl(MethodImplOptions.AggressiveInlining)]
41           protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
             ↪   rootSource, TLink rootTarget);
42
43           [MethodImpl(MethodImplOptions.AggressiveInlining)]
44           protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
             ↪   AsRef<LinksHeader<TLink>>(Header);
45
46           [MethodImpl(MethodImplOptions.AggressiveInlining)]
47           protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
             ↪   AsRef<RawLink<TLink>>(Links + RawLink<TLink>.SizeInBytes *
             ↪   _addressToInt64Converter.Convert(link));
48
49           [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
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
```

```csharp
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             }
```

```csharp
199                     else //if (linkBasePart == @base)
200                     {
201                         if (AreEqual(handler(GetLinkValues(link)), @break))
202                         {
203                             return @break;
204                         }
205                         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
206                         {
207                             return @break;
208                         }
209                         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
210                         {
211                             return @break;
212                         }
213                     }
214                     return @continue;
215                 }
216
217                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
218                 protected override void PrintNodeValue(TLink node, StringBuilder sb)
219                 {
220                     ref var link = ref GetLinkReference(node);
221                     sb.Append(' ');
222                     sb.Append(link.Source);
223                     sb.Append('-');
224                     sb.Append('>');
225                     sb.Append(link.Target);
226                 }
227             }
228         }
```

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

```csharp
1    using System.Runtime.CompilerServices;
2
3    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5    namespace Platform.Data.Doublets.Memory.United.Generic
6    {
7        public unsafe class LinksSourcesAvlBalancedTreeMethods<TLink> :
         ↪  LinksAvlBalancedTreeMethodsBase<TLink>
8        {
9            [MethodImpl(MethodImplOptions.AggressiveInlining)]
10           public LinksSourcesAvlBalancedTreeMethods(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).LeftAsSource;
14
15           [MethodImpl(MethodImplOptions.AggressiveInlining)]
16           protected override ref TLink GetRightReference(TLink node) => ref
         ↪  GetLinkReference(node).RightAsSource;
17
18           [MethodImpl(MethodImplOptions.AggressiveInlining)]
19           protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
20
21           [MethodImpl(MethodImplOptions.AggressiveInlining)]
22           protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
23
24           [MethodImpl(MethodImplOptions.AggressiveInlining)]
25           protected override void SetLeft(TLink node, TLink left) =>
         ↪  GetLinkReference(node).LeftAsSource = left;
26
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) =>
         ↪  GetSizeValue(GetLinkReference(node).SizeAsSource);
32
33           [MethodImpl(MethodImplOptions.AggressiveInlining)]
34           protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
         ↪  GetLinkReference(node).SizeAsSource, size);
35
36           [MethodImpl(MethodImplOptions.AggressiveInlining)]
37           protected override bool GetLeftIsChild(TLink node) =>
         ↪  GetLeftIsChildValue(GetLinkReference(node).SizeAsSource);
38
```

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

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

```csharp
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.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;
```

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

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

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

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

1.45 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsAvlBalancedTreeMethods.cs

```csharp
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
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  }
```

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

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

```csharp
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) => GetLinkReference(node).SizeAsTarget;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override void SetSize(TLink node, TLink size) =>
     ↪   GetLinkReference(node).SizeAsTarget = size;
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
     ↪   TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
     ↪   AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource);
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
     ↪   TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
     ↪   AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource);
47
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          protected override void ClearNode(TLink node)
50          {
51              ref var link = ref GetLinkReference(node);
52              link.LeftAsTarget = Zero;
53              link.RightAsTarget = Zero;
54              link.SizeAsTarget = Zero;
55          }
56      }
57  }
```

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

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

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
         ↪  DefaultLinksSizeStep) { }

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref RawLink<TLink> GetLinkReference(TLink linkIndex) => ref
         ↪  AsRef<RawLink<TLink>>(_links + LinkSizeInBytes * ConvertToInt64(linkIndex));
    }
}
```

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

```csharp
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Disposables;
using Platform.Singletons;
using Platform.Converters;
using Platform.Numbers;
using Platform.Memory;
using Platform.Data.Exceptions;

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

namespace Platform.Data.Doublets.Memory.United.Generic
{
    public abstract class UnitedMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
         ↪  EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
```

```csharp
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
        ↪  UncheckedConverter<TLink, long>.Default;
        private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
        ↪  UncheckedConverter<long, TLink>.Default;

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

        /// <summary>Возвращает размер одной связи в байтах.</summary>
        /// <remarks>
        /// Используется только во вне класса, не рекомедуется использовать внутри.
        /// Так как во вне не обязательно будет доступен unsafe C#.
        /// </remarks>
        public static readonly long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;

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

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

        protected readonly IResizableDirectMemory _memory;
        protected readonly long _memoryReservationStep;

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
        {
            if (memory.ReservedCapacity < memoryReservationStep)
            {
                memory.ReservedCapacity = memoryReservationStep;
            }
            SetPointers(memory);
            ref var header = ref GetHeaderReference();
            // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
            memory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) * LinkSizeInBytes +
            ↪  LinkHeaderSizeInBytes;
            // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
            header.ReservedLinks = ConvertToAddress((memory.ReservedCapacity -
            ↪  LinkHeaderSizeInBytes) / LinkSizeInBytes);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        public virtual TLink Count(IList<TLink> restrictions)
        {
            // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
            if (restrictions.Count == 0)
            {
                return Total;
            }
            var constants = Constants;
            var any = constants.Any;
            var index = restrictions[constants.IndexPart];
            if (restrictions.Count == 1)
            {
                if (AreEqual(index, any))
                {
                    return Total;
                }
                return Exists(index) ? GetOne() : GetZero();
            }
            if (restrictions.Count == 2)
            {
                var value = restrictions[1];
                if (AreEqual(index, any))
                {
                    if (AreEqual(value, any))
                    {
                        return Total; // Any - как отсутствие ограничения
                    }
                    return Add(SourcesTreeMethods.CountUsages(value),
                    ↪  TargetsTreeMethods.CountUsages(value));
                }
                else
                {
                    if (!Exists(index))
                    {
                        return GetZero();
                    }
                    if (AreEqual(value, any))
                    {
                        return GetOne();
                    }
                    ref var storedLinkValue = ref GetLinkReference(index);
                    if (AreEqual(storedLinkValue.Source, value) ||
                    ↪  AreEqual(storedLinkValue.Target, value))
                    {
                        return GetOne();
                    }
                    return GetZero();
                }
            }
            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 Total;
                    }
                    else if (AreEqual(source, any))
                    {
                        return TargetsTreeMethods.CountUsages(target);
                    }
                    else if (AreEqual(target, any))
                    {
                        return SourcesTreeMethods.CountUsages(source);
                    }
                    else //if(source != Any && target != Any)
                    {
                        // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
                        var link = SourcesTreeMethods.Search(source, target);
                        return AreEqual(link, constants.Null) ? GetZero() : GetOne();
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
```

```csharp
                            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>());
                    }
```

```
240                     if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
241                     {
242                         return @break;
243                     }
244                     return Each(handler, new Link<TLink>(index, any, value));
245                 }
246                 else
247                 {
248                     if (!Exists(index))
249                     {
250                         return @continue;
251                     }
252                     if (AreEqual(value, any))
253                     {
254                         return handler(GetLinkStruct(index));
255                     }
256                     ref var storedLinkValue = ref GetLinkReference(index);
257                     if (AreEqual(storedLinkValue.Source, value) ||
258                         AreEqual(storedLinkValue.Target, value))
259                     {
260                         return handler(GetLinkStruct(index));
261                     }
262                     return @continue;
263                 }
264             }
265             if (restrictions.Count == 3)
266             {
267                 var source = restrictions[constants.SourcePart];
268                 var target = restrictions[constants.TargetPart];
269                 if (AreEqual(index, any))
270                 {
271                     if (AreEqual(source, any) && AreEqual(target, any))
272                     {
273                         return Each(handler, Array.Empty<TLink>());
274                     }
275                     else if (AreEqual(source, any))
276                     {
277                         return TargetsTreeMethods.EachUsage(target, handler);
278                     }
279                     else if (AreEqual(target, any))
280                     {
281                         return SourcesTreeMethods.EachUsage(source, handler);
282                     }
283                     else //if(source != Any && target != Any)
284                     {
285                         var link = SourcesTreeMethods.Search(source, target);
286                         return AreEqual(link, constants.Null) ? @continue :
                             ↪   handler(GetLinkStruct(link));
287                     }
288                 }
289                 else
290                 {
291                     if (!Exists(index))
292                     {
293                         return @continue;
294                     }
295                     if (AreEqual(source, any) && AreEqual(target, any))
296                     {
297                         return handler(GetLinkStruct(index));
298                     }
299                     ref var storedLinkValue = ref GetLinkReference(index);
300                     if (!AreEqual(source, any) && !AreEqual(target, any))
301                     {
302                         if (AreEqual(storedLinkValue.Source, source) &&
303                             AreEqual(storedLinkValue.Target, target))
304                         {
305                             return handler(GetLinkStruct(index));
306                         }
307                         return @continue;
308                     }
309                     var value = default(TLink);
310                     if (AreEqual(source, any))
311                     {
312                         value = target;
313                     }
314                     if (AreEqual(target, any))
315                     {
316                         value = source;
```

```csharp
                    }
                    if (AreEqual(storedLinkValue.Source, value) ||
                        AreEqual(storedLinkValue.Target, value))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return @continue;
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не
                поддерживаются.");
        }

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

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

```csharp
                    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);
```

```csharp
            }
            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
    }
}
```

```csharp
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;
    }
}
```

```csharp
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
```

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

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

```csharp
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;
```

```csharp
                                                                                     32
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           33
        protected override bool GreaterThan(ulong first, ulong second) => first > second;  34
                                                                                     35
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           36
        protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;  37
                                                                                     38
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           39
        protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is  40
        ↪  always true for ulong
                                                                                     41
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           42
        protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is  43
        ↪  always >= 0 for ulong
                                                                                     44
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           45
        protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;  46
                                                                                     47
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           48
        protected override bool LessThanZero(ulong value) => false; // value < 0 is always false  49
        ↪  for ulong
                                                                                     50
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           51
        protected override bool LessThan(ulong first, ulong second) => first < second;  52
                                                                                     53
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           54
        protected override ulong Increment(ulong value) => ++value;                  55
                                                                                     56
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           57
        protected override ulong Decrement(ulong value) => --value;                  58
                                                                                     59
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           60
        protected override ulong Add(ulong first, ulong second) => first + second;   61
                                                                                     62
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           63
        protected override ulong Subtract(ulong first, ulong second) => first - second;  64
                                                                                     65
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           66
        protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)  67
        {                                                                            68
            ref var firstLink = ref Links[first];                                    69
            ref var secondLink = ref Links[second];                                  70
            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,      71
            ↪  secondLink.Source, secondLink.Target);
        }                                                                            72
                                                                                     73
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           74
        protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second) 75
        {                                                                            76
            ref var firstLink = ref Links[first];                                    77
            ref var secondLink = ref Links[second];                                  78
            return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,     79
            ↪  secondLink.Source, secondLink.Target);
        }                                                                            80
                                                                                     81
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           82
        protected override ulong GetSizeValue(ulong value) => (value & 4294967264UL) >> 5;  83
                                                                                     84
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           85
        protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =  86
        ↪  storedValue & 31UL | (size & 134217727UL) << 5;
                                                                                     87
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           88
        protected override bool GetLeftIsChildValue(ulong value) => (value & 16UL) >> 4 == 1UL;  89
                                                                                     90
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           91
        protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>  92
        ↪  storedValue = storedValue & 4294967279UL | (As<bool, byte>(ref value) & 1UL) << 4;
                                                                                     93
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           94
        protected override bool GetRightIsChildValue(ulong value) => (value & 8UL) >> 3 == 1UL;  95
                                                                                     96
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           97
        protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>  98
        ↪  storedValue = storedValue & 4294967287UL | (As<bool, byte>(ref value) & 1UL) << 3;
                                                                                     99
        [MethodImpl(MethodImplOptions.AggressiveInlining)]                           100
```

```csharp
101        protected override sbyte GetBalanceValue(ulong value) => unchecked((sbyte)(value & 7UL |
      ↪   0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of
      ↪   sbyte
102
103        [MethodImpl(MethodImplOptions.AggressiveInlining)]
104        protected override void SetBalanceValue(ref ulong storedValue, sbyte value) =>
      ↪   storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
      ↪   value & 3) & 7UL);
105
106        [MethodImpl(MethodImplOptions.AggressiveInlining)]
107        protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
108
109        [MethodImpl(MethodImplOptions.AggressiveInlining)]
110        protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
111    }
112 }
```

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

```csharp
1 using System.Runtime.CompilerServices;
2 using Platform.Data.Doublets.Memory.United.Generic;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Memory.United.Specific
7 {
8     public unsafe abstract class UInt64LinksSizeBalancedTreeMethodsBase :
      ↪   LinksSizeBalancedTreeMethodsBase<ulong>
9     {
10        protected new readonly RawLink<ulong>* Links;
11        protected new readonly LinksHeader<ulong>* Header;
12
13        protected UInt64LinksSizeBalancedTreeMethodsBase(LinksConstants<ulong> constants,
      ↪   RawLink<ulong>* links, LinksHeader<ulong>* header)
14            : base(constants, (byte*)links, (byte*)header)
15        {
16            Links = links;
17            Header = header;
18        }
19
20        [MethodImpl(MethodImplOptions.AggressiveInlining)]
21        protected override ulong GetZero() => 0UL;
22
23        [MethodImpl(MethodImplOptions.AggressiveInlining)]
24        protected override bool EqualToZero(ulong value) => value == 0UL;
25
26        [MethodImpl(MethodImplOptions.AggressiveInlining)]
27        protected override bool AreEqual(ulong first, ulong second) => first == second;
28
29        [MethodImpl(MethodImplOptions.AggressiveInlining)]
30        protected override bool GreaterThanZero(ulong value) => value > 0UL;
31
32        [MethodImpl(MethodImplOptions.AggressiveInlining)]
33        protected override bool GreaterThan(ulong first, ulong second) => first > second;
34
35        [MethodImpl(MethodImplOptions.AggressiveInlining)]
36        protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
37
38        [MethodImpl(MethodImplOptions.AggressiveInlining)]
39        protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
      ↪   always true for ulong
40
41        [MethodImpl(MethodImplOptions.AggressiveInlining)]
42        protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
      ↪   always >= 0 for ulong
43
44        [MethodImpl(MethodImplOptions.AggressiveInlining)]
45        protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
46
47        [MethodImpl(MethodImplOptions.AggressiveInlining)]
48        protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
      ↪   for ulong
49
50        [MethodImpl(MethodImplOptions.AggressiveInlining)]
51        protected override bool LessThan(ulong first, ulong second) => first < second;
52
53        [MethodImpl(MethodImplOptions.AggressiveInlining)]
54        protected override ulong Increment(ulong value) => ++value;
55
56        [MethodImpl(MethodImplOptions.AggressiveInlining)]
57        protected override ulong Decrement(ulong value) => --value;
```

```
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override ulong Add(ulong first, ulong second) => first + second;
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         protected override ulong Subtract(ulong first, ulong second) => first - second;
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
67         {
68             ref var firstLink = ref Links[first];
69             ref var secondLink = ref Links[second];
70             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
               ↪  secondLink.Source, secondLink.Target);
71         }
72
73         [MethodImpl(MethodImplOptions.AggressiveInlining)]
74         protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
75         {
76             ref var firstLink = ref Links[first];
77             ref var secondLink = ref Links[second];
78             return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
               ↪  secondLink.Source, secondLink.Target);
79         }
80
81         [MethodImpl(MethodImplOptions.AggressiveInlining)]
82         protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
83
84         [MethodImpl(MethodImplOptions.AggressiveInlining)]
85         protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
86     }
87 }
```

1.53   ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.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 UInt64LinksSourcesAvlBalancedTreeMethods :
        ↪  UInt64LinksAvlBalancedTreeMethodsBase
8      {
9          public UInt64LinksSourcesAvlBalancedTreeMethods(LinksConstants<ulong> constants,
           ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
           ↪  { }
10
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         protected override ref ulong GetLeftReference(ulong node) => ref
           ↪  Links[node].LeftAsSource;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetRightReference(ulong node) => ref
           ↪  Links[node].RightAsSource;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
           ↪  left;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
           ↪  right;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsSource);
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
           ↪  Links[node].SizeAsSource, size);
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override bool GetLeftIsChild(ulong node) =>
           ↪  GetLeftIsChildValue(Links[node].SizeAsSource);
```

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

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

```csharp
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)]
```

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

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

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

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

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

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

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

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

```csharp
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 UInt64LinksTargetsAvlBalancedTreeMethods :
    ↪  UInt64LinksAvlBalancedTreeMethodsBase
    {
        public UInt64LinksTargetsAvlBalancedTreeMethods(LinksConstants<ulong> constants,
        ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
        ↪  { }

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

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

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

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

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

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

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

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

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

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

                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                protected override ulong GetTreeRoot() => Header->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/UInt64LinksTargetsSizeBalancedTreeMethods.cs

```csharp
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 UInt64LinksTargetsSizeBalancedTreeMethods :
    ↪  UInt64LinksSizeBalancedTreeMethodsBase
    {
```

```csharp
        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;
        }
    }
}
```

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

```csharp
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>
```

```csharp
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, IndexTreeType.Default) { }
38
39              [MethodImpl(MethodImplOptions.AggressiveInlining)]
40              public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
       ↪        memoryReservationStep, LinksConstants<ulong> constants, IndexTreeType indexTreeType)
       ↪        : base(memory, memoryReservationStep, constants)
41              {
42                  if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
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
```

```csharp
        [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;
    }
}
```

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

```csharp
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;
    }
}
```

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

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

```csharp
#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;
        }
    }
}
```

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

```csharp
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))
            {
```

```
36                  throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
37              }
38              var frequency = _frequencyPropertyOperator.Get(link);
39              if (_equalityComparer.Equals(frequency, default))
40              {
41                  return default;
42              }
43              var frequencyNumber = links.GetSource(frequency);
44              return _unaryNumberToAddressConverter.Convert(frequencyNumber);
45          }
46      }
47  }
```

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

```
1  using System.Collections.Generic;
2  using Platform.Exceptions;
3  using Platform.Ranges;
4  using Platform.Converters;
5  using System.Runtime.CompilerServices;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Numbers.Unary
10  {
11      public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<int, TLink>
12      {
13          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
14
15          private readonly TLink[] _unaryNumberPowersOf2;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
19          {
20              _unaryNumberPowersOf2 = new TLink[64];
21              _unaryNumberPowersOf2[0] = one;
22          }
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          public TLink Convert(int power)
26          {
27              Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
                ↪  - 1), nameof(power));
28              if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
29              {
30                  return _unaryNumberPowersOf2[power];
31              }
32              var previousPowerOf2 = Convert(power - 1);
33              var powerOf2 = _links.GetOrCreate(previousPowerOf2, previousPowerOf2);
34              _unaryNumberPowersOf2[power] = powerOf2;
35              return powerOf2;
36          }
37      }
38  }
```

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

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Converters;
4  using Platform.Numbers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Numbers.Unary
9  {
10      public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<TLink>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
13          private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
            ↪  UncheckedConverter<TLink, ulong>.Default;
14          private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
            ↪  UncheckedConverter<ulong, TLink>.Default;
15          private static readonly TLink _zero = default;
16          private static readonly TLink _one = Arithmetic.Increment(_zero);
17
18          private readonly Dictionary<TLink, TLink> _unaryToUInt64;
19          private readonly TLink _unaryOne;
```

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

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

```csharp
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>
```

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

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

```csharp
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)
```

```
19        {
20            var links = _links;
21            var objectProperty = links.SearchOrDefault(@object, property);
22            if (_equalityComparer.Equals(objectProperty, default))
23            {
24                return default;
25            }
26            var constants = links.Constants;
27            var valueLink = links.All(constants.Any, objectProperty).SingleOrDefault();
28            if (valueLink == null)
29            {
30                return default;
31            }
32            return links.GetTarget(valueLink[constants.IndexPart]);
33        }
34
35        [MethodImpl(MethodImplOptions.AggressiveInlining)]
36        public void SetValue(TLink @object, TLink property, TLink value)
37        {
38            var links = _links;
39            var objectProperty = links.GetOrCreate(@object, property);
40            links.DeleteMany(links.AllIndices(links.Constants.Any, objectProperty));
41            links.GetOrCreate(objectProperty, value);
42        }
43    }
44 }
```

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

```
51                    return breakConstant;
52                }
53                return countinueConstant;
54            }, query);
55            return valueContainer;
56        }
57
58        [MethodImpl(MethodImplOptions.AggressiveInlining)]
59        private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
    ↪   ? default : _links.GetTarget(container);
60
61        [MethodImpl(MethodImplOptions.AggressiveInlining)]
62        public void Set(TLink link, TLink value)
63        {
64            var links = _links;
65            var property = links.GetOrCreate(link, _propertyMarker);
66            var container = GetContainer(property);
67            if (_equalityComparer.Equals(container, default))
68            {
69                links.GetOrCreate(property, value);
70            }
71            else
72            {
73                links.Update(container, property, value);
74            }
75        }
76    }
77 }
```

1.66 ./csharp/Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.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.Converters
7  {
8      public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public override TLink Convert(IList<TLink> sequence)
15         {
16             var length = sequence.Count;
17             if (length < 1)
18             {
19                 return default;
20             }
21             if (length == 1)
22             {
23                 return sequence[0];
24             }
25             // Make copy of next layer
26             if (length > 2)
27             {
28                 // TODO: Try to use stackalloc (which at the moment is not working with
    ↪   generics) but will be possible with Sigil
29                 var halvedSequence = new TLink[(length / 2) + (length % 2)];
30                 HalveSequence(halvedSequence, sequence, length);
31                 sequence = halvedSequence;
32                 length = halvedSequence.Length;
33             }
34             // Keep creating layer after layer
35             while (length > 2)
36             {
37                 HalveSequence(sequence, sequence, length);
38                 length = (length / 2) + (length % 2);
39             }
40             return _links.GetOrCreate(sequence[0], sequence[1]);
41         }
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
45         {
46             var loopedLength = length - (length % 2);
47             for (var i = 0; i < loopedLength; i += 2)
48             {
```

```csharp
                    destination[i / 2] = _links.GetOrCreate(source[i], source[i + 1]);
                }
                if (length > loopedLength)
                {
                    destination[length / 2] = source[length - 1];
                }
            }
        }
    }
}
```

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

```csharp
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)
```

```csharp
                    : base(links)
            {
                _baseConverter = baseConverter;
                _doubletFrequenciesCache = doubletFrequenciesCache;
                if (_comparer.Compare(minFrequencyToCompress, _one) < 0)
                {
                    minFrequencyToCompress = _one;
                }
                _minFrequencyToCompress = minFrequencyToCompress;
                _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
                ResetMaxDoublet();
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public override TLink Convert(IList<TLink> source) =>
            ↪   _baseConverter.Convert(Compress(source));

            /// <remarks>
            /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding .
            /// Faster version (doublets' frequencies dictionary is not recreated).
            /// </remarks>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private IList<TLink> Compress(IList<TLink> sequence)
            {
                if (sequence.IsNullOrEmpty())
                {
                    return null;
                }
                if (sequence.Count == 1)
                {
                    return sequence;
                }
                if (sequence.Count == 2)
                {
                    return new[] { _links.GetOrCreate(sequence[0], sequence[1]) };
                }
                // TODO: arraypool with min size (to improve cache locality) or stackallow with Sigil
                var copy = new HalfDoublet[sequence.Count];
                Doublet<TLink> doublet = default;
                for (var i = 1; i < sequence.Count; i++)
                {
                    doublet = new Doublet<TLink>(sequence[i - 1], sequence[i]);
                    LinkFrequency<TLink> data;
                    if (_doInitialFrequenciesIncrement)
                    {
                        data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
                    }
                    else
                    {
                        data = _doubletFrequenciesCache.GetFrequency(ref doublet);
                        if (data == null)
                        {
                            throw new NotSupportedException("If you ask not to increment
                            ↪   frequencies, it is expected that all frequencies for the sequence
                            ↪   are prepared.");
                        }
                    }
                    copy[i - 1].Element = sequence[i - 1];
                    copy[i - 1].DoubletData = data;
                    UpdateMaxDoublet(ref doublet, data);
                }
                copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
                copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
                if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
                {
                    var newLength = ReplaceDoublets(copy);
                    sequence = new TLink[newLength];
                    for (int i = 0; i < newLength; i++)
                    {
                        sequence[i] = copy[i].Element;
                    }
                }
                return sequence;
            }

            /// <remarks>
            /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding
            /// </remarks>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

```
209          {
210              var frequency = data.Frequency;
211              var maxFrequency = _maxDoubletData.Frequency;
212              //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
              ↪   (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
              ↪   compression string data (and gives collisions quickly) */ _maxDoublet.Source +
              ↪   _maxDoublet.Target)))
213              if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
214                  (_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) */
215              {
216                  _maxDoublet = doublet;
217                  _maxDoubletData = data;
218              }
219          }
220      }
221  }
```

1.68 ./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  }
```

1.69 ./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  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
6  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Sequences.Converters
11 {
12     public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
         ↪   EqualityComparer<TLink>.Default;
15         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
16
17         private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
         ↪   sequenceToItsLocalElementLevelsConverter) : base(links)
21             => _sequenceToItsLocalElementLevelsConverter =
             ↪   sequenceToItsLocalElementLevelsConverter;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public OptimalVariantConverter(ILinks<TLink> links, LinkFrequenciesCache<TLink>
         ↪   linkFrequenciesCache)
25             : this(links, new SequenceToItsLocalElementLevelsConverter<TLink>(links, new Frequen⌋
             ↪   ciesCacheBasedLinkToItsFrequencyNumberConverter<TLink>(linkFrequenciesCache))) {
             ↪   }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public OptimalVariantConverter(ILinks<TLink> links)
29             : this(links, new LinkFrequenciesCache<TLink>(links, new
             ↪   TotalSequenceSymbolFrequencyCounter<TLink>(links))) { }
```

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

```csharp
107            [MethodImpl(MethodImplOptions.AggressiveInlining)]
108            private static TLink GetGreatestNeigbourLowerThanCurrentOrCurrent(TLink previous, TLink
    ↪  current, TLink next)
109            {
110                return _comparer.Compare(previous, next) > 0
111                    ? _comparer.Compare(previous, current) < 0 ? previous : current
112                    : _comparer.Compare(next, current) < 0 ? next : current;
113            }
114
115            [MethodImpl(MethodImplOptions.AggressiveInlining)]
116            private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
    ↪  _comparer.Compare(next, current) < 0 ? next : current;
117
118            [MethodImpl(MethodImplOptions.AggressiveInlining)]
119            private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
    ↪  => _comparer.Compare(previous, current) < 0 ? previous : current;
120        }
121  }
```

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

```csharp
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  }
```

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

```csharp
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    }
```

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

```csharp
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>
10      {
11          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
12
13          private readonly ILinks<TLink> _links;
14          private readonly TLink _sequenceMarkerLink;
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
18          {
19              _links = links;
20              _sequenceMarkerLink = sequenceMarkerLink;
21          }
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          public bool IsMatched(TLink sequenceCandidate)
25              => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
26              || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
                ↪ sequenceCandidate), _links.Constants.Null);
27      }
28  }
```

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

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

```
41                  _stack.Push(source);
42                  cursor = target;
43              }
44          }
45          var left = cursor;
46          var right = appendant;
47          while (!_equalityComparer.Equals(cursor = _stack.Pop(), links.Constants.Null))
48          {
49              right = links.GetOrCreate(left, right);
50              left = cursor;
51          }
52          return links.GetOrCreate(left, right);
53      }
54   }
55 }
```

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

```
1  using System.Collections.Generic;
2  using System.Linq;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences
9  {
10     public class DuplicateSegmentsCounter<TLink> : ICounter<int>
11     {
12         private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
            ↪  _duplicateFragmentsProvider;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
            ↪  IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
            ↪  duplicateFragmentsProvider;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
19     }
20 }
```

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

```
1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Interfaces;
6  using Platform.Collections;
7  using Platform.Collections.Lists;
8  using Platform.Collections.Segments;
9  using Platform.Collections.Segments.Walkers;
10 using Platform.Singletons;
11 using Platform.Converters;
12 using Platform.Data.Doublets.Unicode;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     public class DuplicateSegmentsProvider<TLink> :
        ↪  DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
        ↪  IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
19     {
20         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            ↪  UncheckedConverter<TLink, long>.Default;
21         private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
            ↪  UncheckedConverter<TLink, ulong>.Default;
22         private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
            ↪  UncheckedConverter<ulong, TLink>.Default;
23
24         private readonly ILinks<TLink> _links;
25         private readonly ILinks<TLink> _sequences;
26         private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
27         private BitString _visited;
28
29         private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
            ↪  IList<TLink>>>
30         {
31             private readonly IListEqualityComparer<TLink> _listComparer;
32
33             public ItemEquilityComparer() => _listComparer =
                ↪  Default<IListEqualityComparer<TLink>>.Instance;
```

```csharp
34
35                  [MethodImpl(MethodImplOptions.AggressiveInlining)]
36                  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);
37
38                  [MethodImpl(MethodImplOptions.AggressiveInlining)]
39                  public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
                     ↪ (_listComparer.GetHashCode(pair.Key),
                     ↪ _listComparer.GetHashCode(pair.Value)).GetHashCode();
40              }
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
```

```csharp
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));
115            }
116        }
117
118        [MethodImpl(MethodImplOptions.AggressiveInlining)]
119        private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
120        {
121            var duplicates = new List<TLink>();
122            var readAsElement = new HashSet<TLink>();
123            var restrictions = segment.ShiftRight();
124            var constants = _links.Constants;
125            restrictions[0] = constants.Any;
126            _sequences.Each(sequence =>
127            {
128                var sequenceIndex = sequence[constants.IndexPart];
129                duplicates.Add(sequenceIndex);
130                readAsElement.Add(sequenceIndex);
131                return constants.Continue;
132            }, restrictions);
133            if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
134            {
135                return new List<TLink>();
136            }
137            foreach (var duplicate in duplicates)
138            {
139                var duplicateBitIndex = _addressToInt64Converter.Convert(duplicate);
140                _visited.Set(duplicateBitIndex);
141            }
142            if (_sequences is Sequences sequencesExperiments)
143            {
144                var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
   ↪  ashSet<ulong>)(object)readAsElement,
   ↪  (IList<ulong>)segment);
145                foreach (var partiallyMatchedSequence in partiallyMatched)
146                {
147                    var sequenceIndex =
   ↪  _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
148                    duplicates.Add(sequenceIndex);
149                }
150            }
151            duplicates.Sort();
152            return duplicates;
153        }
154
155        [MethodImpl(MethodImplOptions.AggressiveInlining)]
156        private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
157        {
158            if (!(_links is ILinks<ulong> ulongLinks))
159            {
160                return;
161            }
162            var duplicatesKey = duplicatesItem.Key;
163            var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
164            Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
165            var duplicatesList = duplicatesItem.Value;
166            for (int i = 0; i < duplicatesList.Count; i++)
167            {
168                var sequenceIndex = _addressToUInt64Converter.Convert(duplicatesList[i]);
169                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));
170                Console.WriteLine(formatedSequenceStructure);
171                var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
   ↪  ulongLinks);
172                Console.WriteLine(sequenceString);
173            }
```

```
174            Console.WriteLine();
175        }
176    }
177 }
```

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

```csharp
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5  using Platform.Numbers;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
10 {
11     /// <remarks>
12     /// Can be used to operate with many CompressingConverters (to keep global frequencies data
13     ↪  between them).
13     /// TODO: Extract interface to implement frequencies storage inside Links storage
14     /// </remarks>
15     public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
16     {
17         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
18         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
19
20         private static readonly TLink _zero = default;
21         private static readonly TLink _one = Arithmetic.Increment(_zero);
22
23         private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
24         private readonly ICounter<TLink, TLink> _frequencyCounter;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
28             : base(links)
29         {
30             _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
                ↪  DoubletComparer<TLink>.Default);
31             _frequencyCounter = frequencyCounter;
32         }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
36         {
37             var doublet = new Doublet<TLink>(source, target);
38             return GetFrequency(ref doublet);
39         }
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
43         {
44             _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
45             return data;
46         }
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         public void IncrementFrequencies(IList<TLink> sequence)
50         {
51             for (var i = 1; i < sequence.Count; i++)
52             {
53                 IncrementFrequency(sequence[i - 1], sequence[i]);
54             }
55         }
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
59         {
60             var doublet = new Doublet<TLink>(source, target);
61             return IncrementFrequency(ref doublet);
62         }
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         public void PrintFrequencies(IList<TLink> sequence)
66         {
67             for (var i = 1; i < sequence.Count; i++)
68             {
69                 PrintFrequency(sequence[i - 1], sequence[i]);
70             }
71         }
```

```csharp
        [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.");
                //    }
                //}
            }
        }
    }
}
```

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

```csharp
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>
    {
```

```csharp
10          public TLink Frequency { get; set; }
11          public TLink Link { get; set; }
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public LinkFrequency(TLink frequency, TLink link)
15          {
16              Frequency = frequency;
17              Link = link;
18          }
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          public LinkFrequency() { }
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          public void IncrementFrequency() => Frequency = Arithmetic<TLink>.Increment(Frequency);
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          public override string ToString() => $"F: {Frequency}, L: {Link}";
31      }
32  }
```

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

```csharp
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.Sequences.Frequencies.Cache
7   {
8       public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
          ↪  IConverter<Doublet<TLink>, TLink>
9       {
10          private readonly LinkFrequenciesCache<TLink> _cache;
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          public
          ↪  FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
          ↪  cache) => _cache = cache;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
17      }
18  }
```

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

```csharp
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 MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
          ↪  SequenceSymbolFrequencyOneOffCounter<TLink>
9       {
10          private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
          ↪  ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
14              : base(links, sequenceLink, symbol)
15              => _markedSequenceMatcher = markedSequenceMatcher;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          public override TLink Count()
19          {
20              if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
21              {
22                  return default;
23              }
24              return base.Count();
25          }
26      }
27  }
```

```csharp
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4  using Platform.Numbers;
5  using Platform.Data.Sequences;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
10 {
11     public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
12     {
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 }
```

```csharp
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  }
```

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

```
1   using System.Runtime.CompilerServices;
2   using Platform.Interfaces;
3   using Platform.Numbers;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
8   {
9       public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
        ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLink>
10      {
11          private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
            ↪  ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
15              : base(links, symbol)
16              => _markedSequenceMatcher = markedSequenceMatcher;
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          protected override void CountSequenceSymbolFrequency(TLink link)
20          {
21              var symbolFrequencyCounter = new
                ↪  MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
                ↪  _markedSequenceMatcher, link, _symbol);
22              _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
23          }
24      }
25  }
```

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

```
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 TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
9       {
10          private readonly ILinks<TLink> _links;
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public TLink Count(TLink symbol) => new
            ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
17      }
18  }
```

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

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Interfaces;
4   using Platform.Numbers;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
9   {
10      public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
13          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
14
15          protected readonly ILinks<TLink> _links;
16          protected readonly TLink _symbol;
17          protected readonly HashSet<TLink> _visits;
```

```csharp
        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;
        }
    }
}
```

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

```csharp
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)]
```

```csharp
            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;
            }
        }
    }
```

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

```csharp
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;
        }
    }
}
```

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

```csharp
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>
    {
    }
}
```

```csharp
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;

        private readonly LinkFrequenciesCache<TLink> _cache;

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool MightContain(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
            return indexed;
        }

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

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

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

```

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

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

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

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

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

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

namespace Platform.Data.Doublets.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);
    }
}
```

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

```csharp
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.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              {
46                  var indexed = true;
47                  var i = sequence.Count;
48                  while (--i >= 1 && (indexed =
                    ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                    ↪  sequence[i]), default))) { }
49                  return indexed;
50              });
51          }
52      }
53  }
```

1.93 ./csharp/Platform.Data.Doublets/Sequences/Indexes/Unindex.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 Unindex<TLink> : ISequenceIndex<TLink>
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public virtual bool Add(IList<TLink> sequence) => false;
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public virtual bool MightContain(IList<TLink> sequence) => true;
15      }
16  }
```

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

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using System.Linq;
5   using System.Text;
6   using Platform.Collections;
7   using Platform.Collections.Sets;
8   using Platform.Collections.Stacks;
9   using Platform.Data.Exceptions;
10  using Platform.Data.Sequences;
11  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
12  using Platform.Data.Doublets.Sequences.Walkers;
13  using LinkIndex = System.UInt64;
14  using Stack = System.Collections.Generic.Stack<ulong>;
15
16  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
17
18  namespace Platform.Data.Doublets.Sequences
19  {
20      partial class Sequences
21      {
22          #region Create All Variants (Not Practical)
23
24          /// <remarks>
25          /// Number of links that is needed to generate all variants for
26          /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
27          /// </remarks>
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          public ulong[] CreateAllVariants2(ulong[] sequence)
30          {
31              return _sync.ExecuteWriteOperation(() =>
32              {
33                  if (sequence.IsNullOrEmpty())
34                  {
35                      return Array.Empty<ulong>();
36                  }
37                  Links.EnsureLinkExists(sequence);
```

```csharp
38                    if (sequence.Length == 1)
39                    {
40                        return sequence;
41                    }
42                    return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
43                });
44            }
45
46            [MethodImpl(MethodImplOptions.AggressiveInlining)]
47            private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
48            {
49    #if DEBUG
50                if ((stopAt - startAt) < 0)
51                {
52                    throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
                    ↪  меньше или равен stopAt");
53                }
54    #endif
55                if ((stopAt - startAt) == 0)
56                {
57                    return new[] { sequence[startAt] };
58                }
59                if ((stopAt - startAt) == 1)
60                {
61                    return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt]) };
62                }
63                var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
64                var last = 0;
65                for (var splitter = startAt; splitter < stopAt; splitter++)
66                {
67                    var left = CreateAllVariants2Core(sequence, startAt, splitter);
68                    var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
69                    for (var i = 0; i < left.Length; i++)
70                    {
71                        for (var j = 0; j < right.Length; j++)
72                        {
73                            var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
74                            if (variant == Constants.Null)
75                            {
76                                throw new NotImplementedException("Creation cancellation is not
                                ↪  implemented.");
77                            }
78                            variants[last++] = variant;
79                        }
80                    }
81                }
82                return variants;
83            }
84
85            [MethodImpl(MethodImplOptions.AggressiveInlining)]
86            public List<ulong> CreateAllVariants1(params ulong[] sequence)
87            {
88                return _sync.ExecuteWriteOperation(() =>
89                {
90                    if (sequence.IsNullOrEmpty())
91                    {
92                        return new List<ulong>();
93                    }
94                    Links.Unsync.EnsureLinkExists(sequence);
95                    if (sequence.Length == 1)
96                    {
97                        return new List<ulong> { sequence[0] };
98                    }
99                    var results = new
                    ↪  List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
100                   return CreateAllVariants1Core(sequence, results);
101               });
102           }
103
104           [MethodImpl(MethodImplOptions.AggressiveInlining)]
105           private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
106           {
107               if (sequence.Length == 2)
108               {
109                   var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
110                   if (link == Constants.Null)
111                   {
```

```csharp
                            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++)
                            {
```

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

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

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

```
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)
```

```
            {
                Links.Unsync.Each(Constants.Any, right, leftStep =>
                {
                    TryStepLeftUp(handler, left, leftStep);
                    return true;
                });
            }

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

            [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;
```

```
422                         StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
                            ↪  Links.Unsync.GetTarget,
423                             x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                            ↪  x =>
424                             {
425                                 if (filterPosition == sequence.Length)
426                                 {
427                                     filterPosition = -2; // Длиннее чем нужно
428                                     return false;
429                                 }
430                                 if (x != sequence[filterPosition])
431                                 {
432                                     filterPosition = -1;
433                                     return false; // Начинается иначе
434                                 }
435                                 filterPosition++;

437                                 return true;
438                             });
439                         if (filterPosition == sequence.Length)
440                         {
441                             results.Add(resultIndex);
442                         }
443                     }
444                     if (sequence.Length >= 2)
445                     {
446                         StepRight(handler, sequence[0], sequence[1]);
447                     }
448                     var last = sequence.Length - 2;
449                     for (var i = 1; i < last; i++)
450                     {
451                         PartialStepRight(handler, sequence[i], sequence[i + 1]);
452                     }
453                     if (sequence.Length >= 3)
454                     {
455                         StepLeft(handler, sequence[sequence.Length - 2],
                            ↪  sequence[sequence.Length - 1]);
456                     }
457                 }
458                 return results;
459             });
460         }

462         [MethodImpl(MethodImplOptions.AggressiveInlining)]
463         public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
464         {
465             return _sync.ExecuteReadOperation(() =>
466             {
467                 var results = new HashSet<ulong>();
468                 if (sequence.Length > 0)
469                 {
470                     Links.EnsureLinkExists(sequence);
471                     var firstElement = sequence[0];
472                     if (sequence.Length == 1)
473                     {
474                         results.Add(firstElement);
475                         return results;
476                     }
477                     if (sequence.Length == 2)
478                     {
479                         var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
480                         if (doublet != Constants.Null)
481                         {
482                             results.Add(doublet);
483                         }
484                         return results;
485                     }
486                     var matcher = new Matcher(this, sequence, results, null);
487                     if (sequence.Length >= 2)
488                     {
489                         StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
490                     }
491                     var last = sequence.Length - 2;
492                     for (var i = 1; i < last; i++)
493                     {
494                         PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
                            ↪  sequence[i + 1]);
495                     }
```

```
496                    if (sequence.Length >= 3)
497                    {
498                        StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
      ↪    sequence[sequence.Length - 1]);
499                    }
500                }
501                return results;
502            });
503        }
504
505        public const int MaxSequenceFormatSize = 200;
506
507        [MethodImpl(MethodImplOptions.AggressiveInlining)]
508        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);
```

```csharp
            var entered = new HashSet<ulong>();
            var sb = new StringBuilder();
            sb.Append('{');
            if (links.Exists(sequenceLink))
            {
                StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
                    x => linksInSequence.Contains(x) || links.IsFullPoint(x),
                    ↪  entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
                {
                    if (insertComma && sb.Length > 1)
                    {
                        sb.Append(',');
                    }
                    if (entered.Contains(element))
                    {
                        sb.Append('{');
                        elementToString(sb, element);
                        sb.Append('}');
                    }
                    else
                    {
                        elementToString(sb, element);
                    }
                    if (sb.Length < MaxSequenceFormatSize)
                    {
                        return true;
                    }
                    sb.Append(insertComma ? ", ..." : "...");
                    return false;
                });
            }
            sb.Append('}');
            return sb.ToString();
        }

        [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;
```

```csharp
                        }
                    }
                        return true;
                    });
                if (filterPosition == (sequence.Length - 1))
                {
                    filteredResults.Add(result);
                }
            }
                return filteredResults;
            }
            return new List<ulong>();
        });
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            if (sequence.Length > 0)
            {
                Links.EnsureLinkExists(sequence);
                var results = new HashSet<ulong>();
                for (var i = 0; i < sequence.Length; i++)
                {
                    AllUsagesCore(sequence[i], results);
                }
                var filteredResults = new HashSet<ulong>();
                var matcher = new Matcher(this, sequence, filteredResults, null);
                matcher.AddAllPartialMatchedToResults(results);
                return filteredResults;
            }
            return new HashSet<ulong>();
        });
    }

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

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

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

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

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

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

```csharp
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();
```

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(sequence);
                    //var firstElement = sequence[0];
                    //if (sequence.Length == 1)
                    //{
                    //    //results.Add(firstElement);
                    //    return results;
                    //}
                    //if (sequence.Length == 2)
                    //{
                    //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
                    //    //if (doublet != Doublets.Links.Null)
                    //    //    results.Add(doublet);
                    //    return results;
                    //}
                    //var lastElement = sequence[sequence.Length - 1];
                    //Func<ulong, bool> handler = x =>
                    //{
                    //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
                    ↪  results.Add(x);
                    //    return true;
                    //};
                    //if (sequence.Length >= 2)
                    //    StepRight(handler, sequence[0], sequence[1]);
                    //var last = sequence.Length - 2;
                    //for (var i = 1; i < last; i++)
                    //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
                    //if (sequence.Length >= 3)
                    //    StepLeft(handler, sequence[sequence.Length - 2],
                    ↪  sequence[sequence.Length - 1]);
                    //////if (sequence.Length == 1)
                    //////{
                    //////    throw new NotImplementedException(); // all sequences, containing
                    ↪  this element?
                    //////}
                    //////if (sequence.Length == 2)
                    //////{
                    //////    var results = new List<ulong>();
                    //////    PartialStepRight(results.Add, sequence[0], sequence[1]);
                    //////    return results;
                    //////}
                    //////var matches = new List<List<ulong>>();
                    //////var last = sequence.Length - 1;
```

```csharp
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         {
```

```csharp
                            AllUsagesCore(doublet, usages);
                    }
                    return true;
                }
                Links.Unsync.Each(link, Constants.Any, handler);
                Links.Unsync.Each(Constants.Any, link, handler);
            }

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

            [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;
```

```csharp
            }
            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) ||
```

```
1081                    return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
       ↪      link;
1082                }
1083
1084            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1085            private bool CalculateCore(ulong link)
1086            {
1087                // TODO: Проработать защиту от зацикливания
1088                // Основано на SequenceWalker.WalkLeft
1089                Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1090                Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1091                Func<ulong, bool> isElement = IsElement;
1092                void visitLeaf(ulong parent)
1093                {
1094                    if (link != parent)
1095                    {
1096                        _totals[parent]++;
1097                    }
1098                }
1099                void visitNode(ulong parent)
1100                {
1101                    if (link != parent)
1102                    {
1103                        _totals[parent]++;
1104                    }
1105                }
1106                var stack = new Stack();
1107                var element = link;
1108                if (isElement(element))
1109                {
1110                    visitLeaf(element);
1111                }
1112                else
1113                {
1114                    while (true)
1115                    {
1116                        if (isElement(element))
1117                        {
1118                            if (stack.Count == 0)
1119                            {
1120                                break;
1121                            }
1122                            element = stack.Pop();
1123                            var source = getSource(element);
1124                            var target = getTarget(element);
1125                            // Обработка элемента
1126                            if (isElement(target))
1127                            {
1128                                visitLeaf(target);
1129                            }
1130                            if (isElement(source))
1131                            {
1132                                visitLeaf(source);
1133                            }
1134                            element = source;
1135                        }
1136                        else
1137                        {
1138                            stack.Push(element);
1139                            visitNode(element);
1140                            element = getTarget(element);
1141                        }
1142                    }
1143                }
1144                _totals[link]++;
1145                return true;
1146            }
1147        }
1148
1149        private class AllUsagesCollector
1150        {
1151            private readonly ILinks<ulong> _links;
1152            private readonly HashSet<ulong> _usages;
1153
1154            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1155            public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1156            {
1157                _links = links;
1158                _usages = usages;
```

```csharp
            }

            [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;
            }
        }

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

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

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
            {
                _links = links;
                _usages = usages;
            }

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
            ↪  intersectWith, HashSet<ulong> usages)
            {
                _links = links;
                _intersectWith = intersectWith;
                _usages = usages;
                _enter = new HashSet<ulong>(); // защита от зацикливания
            }
```

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

        [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);
```

```csharp
1306                    }
1307                }
1308                if (matchings.Count == 0)
1309                {
1310                    return matchings;
1311                }
1312                return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1313            }
1314
1315            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1316            private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
                ↪ links, params ulong[] sequence)
1317            {
1318                if (sequence == null)
1319                {
1320                    return;
1321                }
1322                for (var i = 0; i < sequence.Length; i++)
1323                {
1324                    if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
                    ↪ !links.Exists(sequence[i]))
1325                    {
1326                        throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                        ↪ $"patternSequence[{i}]");
1327                    }
1328                }
1329            }
1330
1331            // Pattern Matching -> Key To Triggers
1332            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1333            public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1334            {
1335                return _sync.ExecuteReadOperation(() =>
1336                {
1337                    patternSequence = Simplify(patternSequence);
1338                    if (patternSequence.Length > 0)
1339                    {
1340                        EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1341                        var uniqueSequenceElements = new HashSet<ulong>();
1342                        for (var i = 0; i < patternSequence.Length; i++)
1343                        {
1344                            if (patternSequence[i] != Constants.Any && patternSequence[i] !=
                            ↪ ZeroOrMany)
1345                            {
1346                                uniqueSequenceElements.Add(patternSequence[i]);
1347                            }
1348                        }
1349                        var results = new HashSet<ulong>();
1350                        foreach (var uniqueSequenceElement in uniqueSequenceElements)
1351                        {
1352                            AllUsagesCore(uniqueSequenceElement, results);
1353                        }
1354                        var filteredResults = new HashSet<ulong>();
1355                        var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1356                        matcher.AddAllPatternMatchedToResults(results);
1357                        return filteredResults;
1358                    }
1359                    return new HashSet<ulong>();
1360                });
1361            }
1362
1363            // Найти все возможные связи между указанным списком связей.
1364            // Находит связи между всеми указанными связями в любом порядке.
1365            // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
                ↪ несколько раз в последовательности)
1366            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1367            public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1368            {
1369                return _sync.ExecuteReadOperation(() =>
1370                {
1371                    var results = new HashSet<ulong>();
1372                    if (linksToConnect.Length > 0)
1373                    {
1374                        Links.EnsureLinkExists(linksToConnect);
1375                        AllUsagesCore(linksToConnect[0], results);
1376                        for (var i = 1; i < linksToConnect.Length; i++)
1377                        {
1378                            var next = new HashSet<ulong>();
```

```csharp
                        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);
                    }
```

```csharp
                }
                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
```

```csharp
        //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)
```

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

```csharp
                    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));
```

```csharp
                    _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
```

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

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

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

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

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

```csharp
    //            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;
    }
```

```
1997              //}
1998              ////////
1999              //if (_filterPosition == _patternSequence.Length)
2000              //{
2001              //    _filterPosition = -2; // Длиннее чем нужно
2002              //    return false;
2003              //}
2004              //if (element != _patternSequence[_filterPosition])
2005              //{
2006              //    _filterPosition = -1;
2007              //    return false; // Начинается иначе
2008              //}
2009              //_filterPosition++;
2010              //if (_filterPosition == (_patternSequence.Length - 1))
2011              //    return false;
2012              //if (_filterPosition >= 0)
2013              //{
2014              //    if (element == _patternSequence[_filterPosition + 1])
2015              //        _filterPosition++;
2016              //    else
2017              //        return false;
2018              //}
2019              //if (_filterPosition < 0)
2020              //{
2021              //    if (element == _patternSequence[0])
2022              //        _filterPosition = 0;
2023              //}
2024          }

2025
2026          [MethodImpl(MethodImplOptions.AggressiveInlining)]
2027          public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
2028          {
2029              foreach (var sequenceToMatch in sequencesToMatch)
2030              {
2031                  if (PatternMatch(sequenceToMatch))
2032                  {
2033                      _results.Add(sequenceToMatch);
2034                  }
2035              }
2036          }
2037      }

2038
2039      #endregion
2040  }
2041 }
```

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

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

```csharp
        ///
        /// Рост последовательности слева и справа.
        /// Поиск со звёздочкой.
        /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
        /// так же проблема может быть решена при реализации дистанционных триггеров.
        /// Нужны ли уникальные указатели вообще?
        /// Что если обращение к информации будет происходить через содержимое всегда?
        ///
        /// Писать тесты.
        ///
        ///
        /// Можно убрать зависимость от конкретной реализации Links,
        /// на зависимость от абстрактного элемента, который может быть представлен несколькими
        ↪   способами.
        ///
        /// Можно ли как-то сделать один общий интерфейс
        ///
        ///
        /// Блокчейн и/или гит для распределённой записи транзакций.
        ///
        /// </remarks>
        public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
        ↪   (после завершения реализации Sequences)
        {
            /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
            ↪   связей.</summary>
            public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;

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

            public LinksConstants<LinkIndex> Constants { get; }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
            {
                Links = links;
                _sync = links.SyncRoot;
                Options = options;
                Options.ValidateOptions();
                Options.InitOptions(Links);
                Constants = links.Constants;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
            ↪   SequencesOptions<LinkIndex>()) { }

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private LinkIndex GetSequenceElements(LinkIndex sequence)
            {
                if (Options.UseSequenceMarker)
                {
                    var linkContents = new Link<ulong>(Links.GetLink(sequence));
```

```csharp
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(() =>
```

```csharp
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();
271                     sequence[0] = link;
272                     return handler(sequence);
273                 }
274                 else if (restrictions.Count == 2)
275                 {
276                     throw new NotImplementedException();
277                 }
278                 else if (restrictions.Count == 3)
279                 {
280                     return Links.Unsync.Each(handler, restrictions);
281                 }
282                 else
283                 {
284                     var sequence = restrictions.SkipFirst();
285                     if (Options.UseIndex && !Options.Index.MightContain(sequence))
286                     {
287                         return Constants.Break;
288                     }
289                     return EachCore(handler, sequence);
290                 }
291             });
292         }
293
294         [MethodImpl(MethodImplOptions.AggressiveInlining)]
295         private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
            ↪   values)
296         {
297             var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
298             // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
                ↪   Id.
299             Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
                ↪   (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
                ↪   matcher.HandleFullMatched;
300             //if (sequence.Length >= 2)
301             if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
302             {
303                 return Constants.Break;
304             }
305             var last = values.Count - 2;
306             for (var i = 1; i < last; i++)
307             {
308                 if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
                    ↪   Constants.Continue)
309                 {
310                     return Constants.Break;
311                 }
312             }
313             if (values.Count >= 3)
314             {
315                 if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
                    ↪   != Constants.Continue)
316                 {
317                     return Constants.Break;
318                 }
319             }
320             return Constants.Continue;
321         }
322
323         [MethodImpl(MethodImplOptions.AggressiveInlining)]
324         private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
            ↪   left, LinkIndex right)
325         {
326             return Links.Unsync.Each(doublet =>
```

```csharp
                {
                    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)]
            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));

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            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));

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
            ↪   left, LinkIndex stepFrom)
            {
                var upStep = stepFrom;
                var firstTarget = Links.Unsync.GetSource(upStep);
                while (firstTarget != left && firstTarget != upStep)
                {
                    upStep = firstTarget;
                    firstTarget = Links.Unsync.GetTarget(upStep);
                }
                if (firstTarget == left)
                {
                    return handler(new LinkAddress<LinkIndex>(stepFrom));
                }
                return Constants.Continue;
            }

            #endregion

            #region Update

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
            {
                var sequence = restrictions.SkipFirst();
                var newSequence = substitution.SkipFirst();
                if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
                {
                    return Constants.Null;
                }
                if (sequence.IsNullOrEmpty())
                {
                    return Create(substitution);
                }
```

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
        {
            if (Options.UseGarbageCollection)
            {
                var sequenceElements = GetSequenceElements(sequence);
                var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
                var sequenceLink = GetSequenceByElements(sequenceElements);
                var newSequenceElements = GetSequenceElements(newSequence);
                var newSequenceLink = GetSequenceByElements(newSequenceElements);
                if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                {
                    if (sequenceLink != Constants.Null)
                    {
                        Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
                    }
                    Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
                }
                ClearGarbage(sequenceElementsContents.Source);
                ClearGarbage(sequenceElementsContents.Target);
            }
            else
            {
                if (Options.UseSequenceMarker)
                {
                    var sequenceElements = GetSequenceElements(sequence);
                    var sequenceLink = GetSequenceByElements(sequenceElements);
                    var newSequenceElements = GetSequenceElements(newSequence);
                    var newSequenceLink = GetSequenceByElements(newSequenceElements);
                    if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                    {
                        if (sequenceLink != Constants.Null)
                        {
                            Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
                        }
                        Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
                    }
```

```
473                 }
474                 else
475                 {
476                     if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
477                     {
478                         Links.Unsync.MergeAndDelete(sequence, newSequence);
479                     }
480                 }
481             }
482         }

484         #endregion

486         #region Delete

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         }

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         }

546         #endregion

548         #region Compactification

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))
```

```csharp
                {
                    return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
                }
                return _links.Constants.Continue;
            }

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
        {
            if (PartialMatch(sequenceToMatch))
            {
                _results.Add(sequenceToMatch);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkIndex HandlePartialMatched(IList<LinkIndex> restrictions)
        {
            var sequenceToMatch = restrictions[_links.Constants.IndexPart];
            if (PartialMatch(sequenceToMatch))
            {
```

```
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);
794                    }
795                }
796            }
797
798            [MethodImpl(MethodImplOptions.AggressiveInlining)]
799            public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
                ↪  sequencesToMatch)
800            {
801                foreach (var sequenceToMatch in sequencesToMatch)
802                {
803                    if (PartialMatch(sequenceToMatch))
804                    {
805                        _readAsElements.Add(sequenceToMatch);
806                        _results.Add(sequenceToMatch);
807                    }
808                }
809            }
810        }
811
812        #endregion
813    }
814 }
```

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

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences
8  {
9      public static class SequencesExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static TLink Create<TLink>(this ILinks<TLink> sequences, IList<TLink[]>
            ↪  groupedSequence)
13         {
14             var finalSequence = new TLink[groupedSequence.Count];
15             for (var i = 0; i < finalSequence.Length; i++)
16             {
17                 var part = groupedSequence[i];
18                 finalSequence[i] = part.Length == 1 ? part[0] :
                    ↪  sequences.Create(part.ShiftRight());
19             }
20             return sequences.Create(finalSequence.ShiftRight());
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
25         {
26             var list = new List<TLink>();
27             var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
28             sequences.Each(filler.AddSkipFirstAndReturnConstant, new
                ↪  LinkAddress<TLink>(sequence));
29             return list;
30         }
31     }
32 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Collections.Stacks;
```

```csharp
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;
```

```csharp
        }

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

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

        public IConverter<IList<TLink>, TLink> LinksToSequenceConverter
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public ISequenceIndex<TLink> Index
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public ISequenceWalker<TLink> Walker
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void InitOptions(ISynchronizedLinks<TLink> links)
        {
            if (UseSequenceMarker)
            {
                if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
                {
                    SequenceMarkerLink = links.CreatePoint();
                }
                else
                {
                    if (!links.Exists(SequenceMarkerLink))
                    {
                        var link = links.CreatePoint();
                        if (!_equalityComparer.Equals(link, SequenceMarkerLink))
                        {
                            throw new InvalidOperationException("Cannot recreate sequence marker
                            ↪  link.");
                        }
                    }
                }
                if (MarkedSequenceMatcher == null)
                {
                    MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
                    ↪  SequenceMarkerLink);
```

```
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                            }
174                            else
175                            {
176                                totalSequenceSymbolFrequencyCounter = new
                                ↪   TotalSequenceSymbolFrequencyCounter<TLink>(links);
177                            }
178                            var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
                            ↪   totalSequenceSymbolFrequencyCounter);
179                            var compressingConverter = new CompressingConverter<TLink>(links,
                            ↪   balancedVariantConverter, doubletFrequenciesCache);
180                            LinksToSequenceConverter = compressingConverter;
181                        }
182                    }
183                    else
184                    {
185                        if (LinksToSequenceConverter == null)
186                        {
187                            LinksToSequenceConverter = balancedVariantConverter;
188                        }
189                    }
190                    if (UseIndex && Index == null)
191                    {
192                        Index = new SequenceIndex<TLink>(links);
193                    }
194                    if (Walker == null)
195                    {
196                        Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
197                    }
198                }
199
200            [MethodImpl(MethodImplOptions.AggressiveInlining)]
201            public void ValidateOptions()
202            {
203                if (UseGarbageCollection && !UseSequenceMarker)
204                {
205                    throw new NotSupportedException("To use garbage collection UseSequenceMarker
                    ↪   option must be on.");
206                }
207            }
208        }
209    }
```

1.98 ./csharp/Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.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.Walkers
7    {
8        public interface ISequenceWalker<TLink>
9        {
10            [MethodImpl(MethodImplOptions.AggressiveInlining)]
11            IEnumerable<TLink> Walk(TLink sequence);
12        }
13    }
```

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

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

```
 9  {
10      public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11      {
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
            ↪  isElement) : base(links, stack, isElement) { }
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
            ↪  links.IsPartialPoint) { }
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          protected override TLink GetNextElementAfterPop(TLink element) =>
            ↪  _links.GetSource(element);
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          protected override TLink GetNextElementAfterPush(TLink element) =>
            ↪  _links.GetTarget(element);
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override IEnumerable<TLink> WalkContents(TLink element)
26          {
27              var links = _links;
28              var parts = links.GetLink(element);
29              var start = links.Constants.SourcePart;
30              for (var i = parts.Count - 1; i >= start; i--)
31              {
32                  var part = parts[i];
33                  if (IsElement(part))
34                  {
35                      yield return part;
36                  }
37              }
38          }
39      }
40  }
```

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

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Runtime.CompilerServices;
 4
 5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 6
 7  //#define USEARRAYPOOL
 8  #if USEARRAYPOOL
 9  using Platform.Collections;
10  #endif
11
12  namespace Platform.Data.Doublets.Sequences.Walkers
13  {
14      public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
15      {
16          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
17
18          private readonly Func<TLink, bool> _isElement;
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
            ↪  base(links) => _isElement = isElement;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
            ↪  _links.IsPartialPoint;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          public TLink[] ToArray(TLink sequence)
31          {
32              var length = 1;
33              var array = new TLink[length];
34              array[0] = sequence;
35              if (_isElement(sequence))
36              {
37                  return array;
38              }
39              bool hasElements;
```

```csharp
                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++)
            {
```

```
119             if (!_equalityComparer.Equals(array[i], default))
120             {
121                 count++;
122             }
123         }
124         return count;
125     }
126 }
127 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Stacks;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.Walkers
9  {
10     public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
          ↪  isElement) : base(links, stack, isElement) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
          ↪  stack, links.IsPartialPoint) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override TLink GetNextElementAfterPop(TLink element) =>
          ↪  _links.GetTarget(element);
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override TLink GetNextElementAfterPush(TLink element) =>
          ↪  _links.GetSource(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override IEnumerable<TLink> WalkContents(TLink element)
26         {
27             var parts = _links.GetLink(element);
28             for (var i = _links.Constants.SourcePart; i < parts.Count; i++)
29             {
30                 var part = parts[i];
31                 if (IsElement(part))
32                 {
33                     yield return part;
34                 }
35             }
36         }
37     }
38 }
```

1.102 ./csharp/Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Stacks;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.Walkers
9  {
10     public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
          ↪  ISequenceWalker<TLink>
11     {
12         private readonly IStack<TLink> _stack;
13         private readonly Func<TLink, bool> _isElement;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
          ↪  isElement) : base(links)
17         {
18             _stack = stack;
19             _isElement = isElement;
20         }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
            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);
    }
}
```

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

```csharp
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);
```

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

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

```csharp
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;
        }
    }
}
```

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

```csharp
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;
        }
```

```csharp
        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)
        {
            SyncRoot = synchronization;
            Sync = this;
            Unsync = links;
            Constants = links.Constants;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Count(IList<TLinkAddress> restriction) =>
        ↪ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
        ↪ IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
        ↪ restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
        ↪ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
        ↪ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
        ↪ Unsync.Update);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Delete(IList<TLinkAddress> restrictions) =>
        ↪ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
        {
```

```csharp
            if (sequence == null)
            {
                return false;
            }
            var constants = links.Constants;
            for (var i = 0; i < sequence.Length; i++)
            {
                if (sequence[i] == constants.Any)
                {
                    return true;
                }
            }
            return false;
        }

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

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

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

```
 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                         {
103                             appendElement(sb, target);
104                         }
105                         else
106                         {
107                             links.AppendStructure(sb, visited, target.Index, isElement,
                               ↪    appendElement, renderIndex);
108                         }
109                     }
110                     sb.Append(')');
111                 }
112                 else
113                 {
114                     if (renderDebug)
115                     {
116                         sb.Append('*');
117                     }
118                     sb.Append(linkIndex);
119                 }
120             }
121             else
122             {
123                 if (renderDebug)
124                 {
125                     sb.Append('~');
126                 }
127                 sb.Append(linkIndex);
128             }
129         }
130     }
131 }
```

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

```csharp
 1  using System;
 2  using System.Linq;
 3  using System.Collections.Generic;
 4  using System.IO;
 5  using System.Runtime.CompilerServices;
 6  using System.Threading;
 7  using System.Threading.Tasks;
 8  using Platform.Disposables;
 9  using Platform.Timestamps;
10  using Platform.Unsafe;
11  using Platform.IO;
12  using Platform.Data.Doublets.Decorators;
13  using Platform.Exceptions;
14
15  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
16
17  namespace Platform.Data.Doublets
18  {
19      public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
20      {
21          /// <remarks>
22          /// Альтернативные варианты хранения трансформации (элемента транзакции):
23          ///
24          /// private enum TransitionType
25          /// {
26          ///     Creation,
27          ///     UpdateOf,
28          ///     UpdateTo,
29          ///     Deletion
30          /// }
31          ///
32          /// private struct Transition
33          /// {
```

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

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

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

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
        transactionId, Link<ulong> before) : this(uniqueTimestampFactory, transactionId,
        before, default) { }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
        transactionId) : this(uniqueTimestampFactory, transactionId, default, default) {
        }
```

```csharp
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;
```

```csharp
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             }
187             _layer._lastCommitedTransactionId = _layer._currentTransactionId;
188             IsCommitted = true;
189         }
190
191         [MethodImpl(MethodImplOptions.AggressiveInlining)]
192         private void Revert()
193         {
194             EnsureTransactionAllowsWriteOperations(this);
195             var transitionsToRevert = new Transition[_transitions.Count];
196             _transitions.CopyTo(transitionsToRevert, 0);
197             for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
198             {
199                 _layer.RevertTransition(transitionsToRevert[i]);
200             }
201             IsReverted = true;
202         }
203
204         [MethodImpl(MethodImplOptions.AggressiveInlining)]
205         public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
            ↪   Transaction transaction)
206         {
207             layer._currentTransactionId = layer._lastCommitedTransactionId + 1;
208             layer._currentTransactionTransitions = transaction._transitions;
209             layer._currentTransaction = transaction;
210         }
211
212         [MethodImpl(MethodImplOptions.AggressiveInlining)]
213         public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
214         {
215             if (transaction.IsReverted)
216             {
217                 throw new InvalidOperationException("Transation is reverted.");
218             }
219             if (transaction.IsCommitted)
220             {
221                 throw new InvalidOperationException("Transation is commited.");
222             }
223         }
224
225         [MethodImpl(MethodImplOptions.AggressiveInlining)]
226         protected override void Dispose(bool manual, bool wasDisposed)
227         {
228             if (!wasDisposed && _layer != null && !_layer.Disposable.IsDisposed)
229             {
230                 if (!IsCommitted && !IsReverted)
231                 {
232                     Revert();
233                 }
234                 _layer.ResetCurrentTransation();
235             }
236         }
237     }
238
239     public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
240
241     private readonly string _logAddress;
242     private readonly FileStream _log;
243     private readonly Queue<Transition> _transitions;
244     private readonly UniqueTimestampFactory _uniqueTimestampFactory;
245     private Task _transitionsPusher;
246     private Transition _lastCommitedTransition;
247     private ulong _currentTransactionId;
248     private Queue<Transition> _currentTransactionTransitions;
249     private Transaction _currentTransaction;
250     private ulong _lastCommitedTransactionId;
251
252     [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
253         public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
254             : base(links)
255         {
256             if (string.IsNullOrWhiteSpace(logAddress))
257             {
258                 throw new ArgumentNullException(nameof(logAddress));
259             }
260             // В первой строке файла хранится последняя закоммиченную транзакцию.
261             // При запуске это используется для проверки удачного закрытия файла лога.
262             // In the first line of the file the last committed transaction is stored.
263             // On startup, this is used to check that the log file is successfully closed.
264             var lastCommitedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
265             var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
266             if (!lastCommitedTransition.Equals(lastWrittenTransition))
267             {
268                 Dispose();
269                 throw new NotSupportedException("Database is damaged, autorecovery is not
    ↪   supported yet.");
270             }
271             if (lastCommitedTransition == default)
272             {
273                 FileHelpers.WriteFirst(logAddress, lastCommitedTransition);
274             }
275             _lastCommitedTransition = lastCommitedTransition;
276             // TODO: Think about a better way to calculate or store this value
277             var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
278             _lastCommitedTransactionId = allTransitions.Length > 0 ? allTransitions.Max(x =>
    ↪   x.TransactionId) : 0;
279             _uniqueTimestampFactory = new UniqueTimestampFactory();
280             _logAddress = logAddress;
281             _log = FileHelpers.Append(logAddress);
282             _transitions = new Queue<Transition>();
283             _transitionsPusher = new Task(TransitionsPusher);
284             _transitionsPusher.Start();
285         }
286
287         [MethodImpl(MethodImplOptions.AggressiveInlining)]
288         public IList<ulong> GetLinkValue(ulong link) => _links.GetLink(link);
289
290         [MethodImpl(MethodImplOptions.AggressiveInlining)]
291         public override ulong Create(IList<ulong> restrictions)
292         {
293             var createdLinkIndex = _links.Create();
294             var createdLink = new Link<ulong>(_links.GetLink(createdLinkIndex));
295             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
    ↪   default, createdLink));
296             return createdLinkIndex;
297         }
298
299         [MethodImpl(MethodImplOptions.AggressiveInlining)]
300         public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
301         {
302             var linkIndex = restrictions[_constants.IndexPart];
303             var beforeLink = new Link<ulong>(_links.GetLink(linkIndex));
304             linkIndex = _links.Update(restrictions, substitution);
305             var afterLink = new Link<ulong>(_links.GetLink(linkIndex));
306             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
    ↪   beforeLink, afterLink));
307             return linkIndex;
308         }
309
310         [MethodImpl(MethodImplOptions.AggressiveInlining)]
311         public override void Delete(IList<ulong> restrictions)
312         {
313             var link = restrictions[_constants.IndexPart];
314             var deletedLink = new Link<ulong>(_links.GetLink(link));
315             _links.Delete(link);
316             CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
    ↪   deletedLink, default));
317         }
318
319         [MethodImpl(MethodImplOptions.AggressiveInlining)]
320         private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
    ↪   _transitions;
321
322         [MethodImpl(MethodImplOptions.AggressiveInlining)]
323         private void CommitTransition(Transition transition)
324         {
```

```csharp
            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();
```

```
403                    FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
404                }
405                catch (Exception ex)
406                {
407                    ex.Ignore();
408                }
409            }
410
411            #region DisposalBase
412
413            [MethodImpl(MethodImplOptions.AggressiveInlining)]
414            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    }
```

1.108 ./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  }
```

1.109 ./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<string, IList<TLink>> _stringToUnicodeSymbolListConverter;
13          private readonly IConverter<IList<TLink>, TLink> _unicodeSymbolListToSequenceConverter;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
         ↪  IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
         ↪  unicodeSymbolListToSequenceConverter) : base(links)
17          {
```

```
18              _stringToUnicodeSymbolListConverter = stringToUnicodeSymbolListConverter;
19              _unicodeSymbolListToSequenceConverter = unicodeSymbolListToSequenceConverter;
20          }
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
            ↪  IList<TLink>> stringToUnicodeSymbolListConverter, ISequenceIndex<TLink> index,
            ↪  IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
            ↪  unicodeSequenceMarker)
24              : this(links, stringToUnicodeSymbolListConverter, new
                ↪  UnicodeSymbolsListToUnicodeSequenceConverter<TLink>(links, index,
                ↪  listToSequenceLinkConverter, unicodeSequenceMarker)) { }
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
            ↪  charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
            ↪  TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker)
28              : this(links, new
                ↪  StringToUnicodeSymbolsListConverter<TLink>(charToUnicodeSymbolConverter), index,
                ↪  listToSequenceLinkConverter, unicodeSequenceMarker) { }
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
            ↪  charToUnicodeSymbolConverter, IConverter<IList<TLink>, TLink>
            ↪  listToSequenceLinkConverter, TLink unicodeSequenceMarker)
32              : this(links, charToUnicodeSymbolConverter, new Unindex<TLink>(),
                ↪  listToSequenceLinkConverter, unicodeSequenceMarker) { }
33
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
            ↪  IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
            ↪  listToSequenceLinkConverter, TLink unicodeSequenceMarker)
36              : this(links, stringToUnicodeSymbolListConverter, new Unindex<TLink>(),
                ↪  listToSequenceLinkConverter, unicodeSequenceMarker) { }
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          public TLink Convert(string source)
40          {
41              var elements = _stringToUnicodeSymbolListConverter.Convert(source);
42              return _unicodeSymbolListToSequenceConverter.Convert(elements);
43          }
44      }
45  }
```

1.110 ./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSymbolsListConverter.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.Unicode
8  {
9      public class StringToUnicodeSymbolsListConverter<TLink> : IConverter<string, IList<TLink>>
10     {
11         private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public StringToUnicodeSymbolsListConverter(IConverter<char, TLink>
           ↪  charToUnicodeSymbolConverter) => _charToUnicodeSymbolConverter =
           ↪  charToUnicodeSymbolConverter;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public IList<TLink> Convert(string source)
18         {
19             var elements = new TLink[source.Length];
20             for (var i = 0; i < elements.Length; i++)
21             {
22                 elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
23             }
24             return elements;
25         }
26     }
27  }
```

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

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

```csharp
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;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static char FromLinkToChar(ulong link) => (char)(link - 1);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsCharLink(ulong link) => link <= MapSize;

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

```csharp
                }
                return sb.ToString();
            }

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

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

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

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

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

```csharp
157              [MethodImpl(MethodImplOptions.AggressiveInlining)]
158              public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
159              {
160                  var result = new List<ulong[]>();
161                  var offset = 0;
162                  while (offset < array.Length)
163                  {
164                      var relativeLength = 1;
165                      if (array[offset] <= LastCharLink)
166                      {
167                          var currentCategory =
                              ↪  CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
168                          var absoluteLength = offset + relativeLength;
169                          while (absoluteLength < array.Length &&
170                                  array[absoluteLength] <= LastCharLink &&
171                                  currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
                                  ↪  array[absoluteLength])))
172                          {
173                              relativeLength++;
174                              absoluteLength++;
175                          }
176                      }
177                      else
178                      {
179                          var absoluteLength = offset + relativeLength;
180                          while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
181                          {
182                              relativeLength++;
183                              absoluteLength++;
184                          }
185                      }
186                      // copy array
187                      var innerSequence = new ulong[relativeLength];
188                      var maxLength = offset + relativeLength;
189                      for (var i = offset; i < maxLength; i++)
190                      {
191                          innerSequence[i - offset] = array[i];
192                      }
193                      result.Add(innerSequence);
194                      offset += relativeLength;
195                  }
196                  return result;
197              }
198          }
199      }
```

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

```csharp
1   using System;
2   using System.Linq;
3   using System.Runtime.CompilerServices;
4   using Platform.Interfaces;
5   using Platform.Converters;
6   using Platform.Data.Doublets.Sequences.Walkers;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets.Unicode
11  {
12      public class UnicodeSequenceToStringConverter<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<TLink, string>
13      {
14          private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
15          private readonly ISequenceWalker<TLink> _sequenceWalker;
16          private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
            ↪  unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
            ↪  IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
20          {
21              _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
22              _sequenceWalker = sequenceWalker;
23              _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
24          }
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          public string Convert(TLink source)
28          {
29              if (!_unicodeSequenceCriterionMatcher.IsMatched(source))
30              {
```

```
31                    throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
   ↪    not a unicode sequence.");
32                }
33                var sequence = _links.GetSource(source);
34                var charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.
   ↪    Convert).ToArray();
35                return new string(charArray);
36            }
37        }
38    }
```

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

```
1    using System;
2    using System.Runtime.CompilerServices;
3    using Platform.Interfaces;
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.Unicode
9    {
10       public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
   ↪    IConverter<TLink, char>
11       {
12           private static readonly UncheckedConverter<TLink, char> _addressToCharConverter =
   ↪    UncheckedConverter<TLink, char>.Default;
13
14           private readonly IConverter<TLink> _numberToAddressConverter;
15           private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;
16
17           [MethodImpl(MethodImplOptions.AggressiveInlining)]
18           public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
   ↪    numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
   ↪    base(links)
19           {
20               _numberToAddressConverter = numberToAddressConverter;
21               _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
22           }
23
24           [MethodImpl(MethodImplOptions.AggressiveInlining)]
25           public char Convert(TLink source)
26           {
27               if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
28               {
29                   throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
   ↪    not a unicode symbol.");
30               }
31               return _addressToCharConverter.Convert(_numberToAddressConverter.Convert(_links.GetS
   ↪    ource(source)));
32           }
33       }
34   }
```

1.114   ./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.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 UnicodeSymbolsListToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
   ↪    IConverter<IList<TLink>, TLink>
11       {
12           private readonly ISequenceIndex<TLink> _index;
13           private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
14           private readonly TLink _unicodeSequenceMarker;
15
16           [MethodImpl(MethodImplOptions.AggressiveInlining)]
17           public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
   ↪    ISequenceIndex<TLink> index, IConverter<IList<TLink>, TLink>
   ↪    listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
18           {
19               _index = index;
20               _listToSequenceLinkConverter = listToSequenceLinkConverter;
21               _unicodeSequenceMarker = unicodeSequenceMarker;
22           }
```

```
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
       ↪   IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
       ↪   unicodeSequenceMarker)
26              : this(links, new Unindex<TLink>(), listToSequenceLinkConverter,
       ↪   unicodeSequenceMarker) { }
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          public TLink Convert(IList<TLink> list)
30          {
31              _index.Add(list);
32              var sequence = _listToSequenceLinkConverter.Convert(list);
33              return _links.GetOrCreate(sequence, _unicodeSequenceMarker);
34          }
35      }
36  }
```

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

```
1   using System;
2   using Xunit;
3   using Platform.Reflection;
4   using Platform.Memory;
5   using Platform.Scopes;
6   using Platform.Data.Doublets.Memory.United.Generic;
7
8   namespace Platform.Data.Doublets.Tests
9   {
10      public unsafe static class GenericLinksTests
11      {
12          [Fact]
13          public static void CRUDTest()
14          {
15              Using<byte>(links => links.TestCRUDOperations());
16              Using<ushort>(links => links.TestCRUDOperations());
17              Using<uint>(links => links.TestCRUDOperations());
18              Using<ulong>(links => links.TestCRUDOperations());
19          }
20
21          [Fact]
22          public static void RawNumbersCRUDTest()
23          {
24              Using<byte>(links => links.TestRawNumbersCRUDOperations());
25              Using<ushort>(links => links.TestRawNumbersCRUDOperations());
26              Using<uint>(links => links.TestRawNumbersCRUDOperations());
27              Using<ulong>(links => links.TestRawNumbersCRUDOperations());
28          }
29
30          [Fact]
31          public static void MultipleRandomCreationsAndDeletionsTest()
32          {
33              Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
       ↪   MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
       ↪   implementation of tree cuts out 5 bits from the address space.
34              Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
       ↪   stMultipleRandomCreationsAndDeletions(100));
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  }
```

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

```
1   using Xunit;
2
3   namespace Platform.Data.Doublets.Tests
4   {
```

```csharp
    public class ILinksExtensionsTests
    {
        [Fact]
        public void FormatTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var link = links.Create();
                var linkString = links.Format(link);
                Assert.Equal("(1: 1 1)", linkString);
            }
        }
    }
}
```

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

```csharp
using Xunit;

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

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

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

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

```csharp
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;
using Platform.Data.Doublets.Memory;

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.
```

```
40    Nulla facilisi etiam dignissim diam quis enim lobortis scelerisque.
41    Imperdiet proin fermentum leo vel orci.
42    In ante metus dictum at tempor commodo.
43    Nisi lacus sed viverra tellus in.
44    Quam vulputate dignissim suspendisse in.
45    Elit scelerisque mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus.
46    Gravida cum sociis natoque penatibus et magnis dis parturient.
47    Risus quis varius quam quisque id diam.
48    Congue nisi vitae suscipit tellus mauris a diam maecenas.
49    Eget nunc scelerisque viverra mauris in aliquam sem fringilla.
50    Pharetra vel turpis nunc eget lorem dolor sed viverra.
51    Mattis pellentesque id nibh tortor id aliquet.
52    Purus non enim praesent elementum facilisis leo vel.
53    Etiam sit amet nisl purus in mollis nunc sed.
54    Tortor at auctor urna nunc id cursus metus aliquam.
55    Volutpat odio facilisis mauris sit amet.
56    Turpis egestas pretium aenean pharetra magna ac placerat.
57    Fermentum dui faucibus in ornare quam viverra orci sagittis eu.
58    Porttitor leo a diam sollicitudin tempor id eu.
59    Volutpat sed cras ornare arcu dui.
60    Ut aliquam purus sit amet luctus venenatis lectus magna.
61    Aliquet risus feugiat in ante metus dictum at.
62    Mattis nunc sed blandit libero.
63    Elit pellentesque habitant morbi tristique senectus et netus.
64    Nibh sit amet commodo nulla facilisi nullam vehicula ipsum a.
65    Enim sit amet venenatis urna cursus eget nunc scelerisque viverra.
66    Amet venenatis urna cursus eget nunc scelerisque viverra mauris in.
67    Diam donec adipiscing tristique risus nec feugiat.
68    Pulvinar mattis nunc sed blandit libero volutpat.
69    Cras fermentum odio eu feugiat pretium nibh ipsum.
70    In nulla posuere sollicitudin aliquam ultrices sagittis orci a.
71    Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.
72    A iaculis at erat pellentesque.
73    Morbi blandit cursus risus at ultrices mi tempus imperdiet nulla.
74    Eget lorem dolor sed viverra ipsum nunc.
75    Leo a diam sollicitudin tempor id eu.
76    Interdum consectetur libero id faucibus nisl tincidunt eget nullam non.";
77
78            [Fact]
79            public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
80            {
81                using (var scope = new TempLinksTestScope(useSequences: false))
82                {
83                    var links = scope.Links;
84                    var constants = links.Constants;
85
86                    links.UseUnicode();
87
88                    var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
89
90                    var meaningRoot = links.CreatePoint();
91                    var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
92                    var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
93                    var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                    ↪  constants.Itself);
94
95                    var unaryNumberToAddressConverter = new
                    ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
96                    var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
97                    var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
98                    var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                    ↪  frequencyPropertyMarker, frequencyMarker);
99                    var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
100                   var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
101                   var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
102                   var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
103
104                   var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
                    ↪  Walker = new LeveledSequenceWalker<ulong>(links) });
105
106                   ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
                    ↪  index, optimalVariantConverter);
107                }
108           }
```

```csharp
109
110          [Fact]
111          public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
112          {
113              using (var scope = new TempLinksTestScope(useSequences: false))
114              {
115                  var links = scope.Links;
116
117                  links.UseUnicode();
118
119                  var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
120
121                  var totalSequenceSymbolFrequencyCounter = new
                     ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links);
122
123                  var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
                     ↪  totalSequenceSymbolFrequencyCounter);
124
125                  var index = new
                     ↪  CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
126                  var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque ⌐
                     ↪  ncyNumberConverter<ulong>(linkFrequenciesCache);
127
128                  var sequenceToItsLocalElementLevelsConverter = new
                     ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                     ↪  linkToItsFrequencyNumberConverter);
129                  var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                     ↪  sequenceToItsLocalElementLevelsConverter);
130
131                  var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
                     ↪  Walker = new LeveledSequenceWalker<ulong>(links) });
132
133                  ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
                     ↪  index, optimalVariantConverter);
134              }
135          }
136
137          private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
             ↪  SequenceToItsLocalElementLevelsConverter<ulong>
             ↪  sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
             ↪  OptimalVariantConverter<ulong> optimalVariantConverter)
138          {
139              index.Add(sequence);
140
141              var optimalVariant = optimalVariantConverter.Convert(sequence);
142
143              var readSequence1 = sequences.ToList(optimalVariant);
144
145              Assert.True(sequence.SequenceEqual(readSequence1));
146          }
147
148          [Fact]
149          public static void SavedSequencesOptimizationTest()
150          {
151              LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
                 ↪  (long.MaxValue + 1UL, ulong.MaxValue));
152
153              using (var memory = new HeapResizableDirectMemory())
154              using (var disposableLinks = new UInt64UnitedMemoryLinks(memory,
                 ↪  UInt64UnitedMemoryLinks.DefaultLinksSizeStep, constants, IndexTreeType.Default))
155              {
156                  var links = new UInt64Links(disposableLinks);
157
158                  var root = links.CreatePoint();
159
160                  //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
161                  var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
162
163                  var unicodeSymbolMarker = links.GetOrCreate(root,
                     ↪  addressToNumberConverter.Convert(1));
164                  var unicodeSequenceMarker = links.GetOrCreate(root,
                     ↪  addressToNumberConverter.Convert(2));
165
166                  var totalSequenceSymbolFrequencyCounter = new
                     ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links);
167                  var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
                     ↪  totalSequenceSymbolFrequencyCounter);
168                  var index = new
                     ↪  CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
```

```
169            var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque⌋
       ↪ ncyNumberConverter<ulong>(linkFrequenciesCache);
170            var sequenceToItsLocalElementLevelsConverter = new
       ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
       ↪ linkToItsFrequencyNumberConverter);
171            var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
       ↪ sequenceToItsLocalElementLevelsConverter);
172
173            var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
       ↪ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
174
175            var unicodeSequencesOptions = new SequencesOptions<ulong>()
176            {
177                UseSequenceMarker = true,
178                SequenceMarkerLink = unicodeSequenceMarker,
179                UseIndex = true,
180                Index = index,
181                LinksToSequenceConverter = optimalVariantConverter,
182                Walker = walker,
183                UseGarbageCollection = true
184            };
185
186            var unicodeSequences = new Sequences.Sequences(new
       ↪ SynchronizedLinks<ulong>(links), unicodeSequencesOptions);
187
188            // Create some sequences
189            var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
       ↪ StringSplitOptions.RemoveEmptyEntries);
190            var arrays = strings.Select(x => x.Select(y =>
       ↪ addressToNumberConverter.Convert(y)).ToArray()).ToArray();
191            for (int i = 0; i < arrays.Length; i++)
192            {
193                unicodeSequences.Create(arrays[i].ShiftRight());
194            }
195
196            var linksCountAfterCreation = links.Count();
197
198            // get list of sequences links
199            // for each sequence link
200            //   create new sequence version
201            //   if new sequence is not the same as sequence link
202            //     delete sequence link
203            //     collect garbadge
204            unicodeSequences.CompactAll();
205
206            var linksCountAfterCompactification = links.Count();
207
208            Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
209        }
210    }
211    }
212 }
```

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

```
 1 using System;
 2 using System.Collections.Generic;
 3 using System.Diagnostics;
 4 using System.Linq;
 5 using Xunit;
 6 using Platform.Data.Sequences;
 7 using Platform.Data.Doublets.Sequences.Converters;
 8 using Platform.Data.Doublets.Sequences.Walkers;
 9 using Platform.Data.Doublets.Sequences;
10
11 namespace Platform.Data.Doublets.Tests
12 {
13    public static class ReadSequenceTests
14    {
15        [Fact]
16        public static void ReadSequenceTest()
17        {
18            const long sequenceLength = 2000;
19
20            using (var scope = new TempLinksTestScope(useSequences: false))
21            {
22                var links = scope.Links;
23                var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong> {
       ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
24
25                var sequence = new ulong[sequenceLength];
```

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

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

                     var sw1 = Stopwatch.StartNew();
                     var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();

                     var sw2 = Stopwatch.StartNew();
                     var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();

                     var sw3 = Stopwatch.StartNew();
                     var readSequence2 = new List<ulong>();
                     SequenceWalker.WalkRight(balancedVariant,
                                              links.GetSource,
                                              links.GetTarget,
                                              links.IsPartialPoint,
                                              readSequence2.Add);
                     sw3.Stop();

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

                     Assert.True(sequence.SequenceEqual(readSequence2));

                     // Assert.True(sw2.Elapsed < sw3.Elapsed);

                     Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
                     ↪   {sw2.Elapsed}");

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

1.120 ./csharp/Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs

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

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

        [Fact]
        public static void BasicFileMappedMemoryTest()
        {
            var tempFilename = Path.GetTempFileName();
            using (var memoryAdapter = new UInt64UnitedMemoryLinks(tempFilename))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
            File.Delete(tempFilename);
        }

        [Fact]
        public static void BasicHeapMemoryTest()
        {
            using (var memory = new
            ↪   HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
            ↪   UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
        }

        private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
        {
```

```csharp
                    var link = memoryAdapter.Create();
                    memoryAdapter.Delete(link);
                }

                [Fact]
                public static void NonexistentReferencesHeapMemoryTest()
                {
                    using (var memory = new
                    ↪    HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
                    using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
                    ↪    UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
                    {
                        memoryAdapter.TestNonexistentReferences();
                    }
                }

                private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
                {
                    var link = memoryAdapter.Create();
                    memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
                    var resultLink = _constants.Null;
                    memoryAdapter.Each(foundLink =>
                    {
                        resultLink = foundLink[_constants.IndexPart];
                        return _constants.Break;
                    }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
                    Assert.True(resultLink == link);
                    Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
                    memoryAdapter.Delete(link);
                }
            }
        }
```

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

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

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

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

```csharp
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()
        //{
```

```csharp
 66        //     var tempFilename = Path.GetTempFileName();
 67
 68        //     const long sequenceLength = 8;
 69
 70        //     const ulong itself = LinksConstants.Itself;
 71
 72        //     using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
         ↪ DefaultLinksSizeStep))
 73        //     using (var links = new Links(memoryAdapter))
 74        //     {
 75        //         var sequence = new ulong[sequenceLength];
 76        //         for (var i = 0; i < sequenceLength; i++)
 77        //             sequence[i] = links.Create(itself, itself);
 78
 79        //         SequencesOptions o = new SequencesOptions();
 80
 81        // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
 82        //         o.
 83
 84        //         var sequences = new Sequences(links);
 85
 86        //         var sw1 = Stopwatch.StartNew();
 87        //         var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
 88
 89        //         var sw2 = Stopwatch.StartNew();
 90        //         var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
 91
 92        //         Assert.True(results1.Count > results2.Length);
 93        //         Assert.True(sw1.Elapsed > sw2.Elapsed);
 94
 95        //         for (var i = 0; i < sequenceLength; i++)
 96        //             links.Delete(sequence[i]);
 97        //     }
 98
 99        //     File.Delete(tempFilename);
100        //}
101
102        [Fact]
103        public static void AllVariantsSearchTest()
104        {
105            const long sequenceLength = 8;
106
107            using (var scope = new TempLinksTestScope(useSequences: true))
108            {
109                var links = scope.Links;
110                var sequences = scope.Sequences;
111
112                var sequence = new ulong[sequenceLength];
113                for (var i = 0; i < sequenceLength; i++)
114                {
115                    sequence[i] = links.Create();
116                }
117
118                var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
119
120                //for (int i = 0; i < createResults.Length; i++)
121                //    sequences.Create(createResults[i]);
122
123                var sw0 = Stopwatch.StartNew();
124                var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
125
126                var sw1 = Stopwatch.StartNew();
127                var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
128
129                var sw2 = Stopwatch.StartNew();
130                var searchResults2 = sequences.Each1(sequence); sw2.Stop();
131
132                var sw3 = Stopwatch.StartNew();
133                var searchResults3 = sequences.Each(sequence.ShiftRight()); sw3.Stop();
134
135                var intersection0 = createResults.Intersect(searchResults0).ToList();
136                Assert.True(intersection0.Count == searchResults0.Count);
137                Assert.True(intersection0.Count == createResults.Length);
138
139                var intersection1 = createResults.Intersect(searchResults1).ToList();
140                Assert.True(intersection1.Count == searchResults1.Count);
141                Assert.True(intersection1.Count == createResults.Length);
142
143                var intersection2 = createResults.Intersect(searchResults2).ToList();
144                Assert.True(intersection2.Count == searchResults2.Count);
```

```csharp
                        Assert.True(intersection2.Count == createResults.Length);

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

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

            [Fact]
            public static void BalancedVariantSearchTest()
            {
                const long sequenceLength = 200;

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

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

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

                    var sw1 = Stopwatch.StartNew();
                    var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();

                    var sw2 = Stopwatch.StartNew();
                    var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();

                    var sw3 = Stopwatch.StartNew();
                    var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();

                    // На количестве в 200 элементов это будет занимать вечность
                    //var sw4 = Stopwatch.StartNew();
                    //var searchResults4 = sequences.Each(sequence); sw4.Stop();

                    Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);

                    Assert.True(searchResults3.Count == 1 && balancedVariant ==
                    ↪   searchResults3.First());

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

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

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

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

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

                    var createResults = sequences.CreateAllVariants2(sequence);

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

```csharp
                    var partialSequence = new ulong[sequenceLength - 2];

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

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

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

                    //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++)
```

```csharp
                        {
                            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);
```

```csharp
374                    Assert.True(index.MightContain(sequence));
375                }
376            }
377        }
378
379        /// <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>
380        private static readonly string _exampleText =
381            @"([english
               ↪ version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
382
383 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?
384
385 [![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)
386
387 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?
388
389 [![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)
390
391 А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть
     ↪ так? Инверсия? Отражение? Сумма?
392
393 [![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)
394
395 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?
396
397 [![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)
398
399 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?
400
401 [![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)
402
403 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?
404
405 [![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)
406
407 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?
408
409 [![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)
410
```

Шаг назад. Возьмём опять отделённую область, которая лишь та же замкнутая линия, что ещё она
может представлять собой? Объект? Но в чём его суть? Разве не в том, что у него есть
граница, разделяющая внутреннее и внешнее? Допустим связь, стрелка, линия соединяет два
объекта, как бы это выглядело?

[![белая связь, чёрная направленная
связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png ""белая
связь, чёрная направленная
связь"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png)

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

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

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

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

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

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

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

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

...

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

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

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

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

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

```
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);
```

```csharp
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]);
```

```csharp
            }

            var elapsed3 = sw3.Elapsed;

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

            // Assert.True(elapsed1 > elapsed2);

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

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

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

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

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

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

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

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

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

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

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

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

            var duplicates1 = duplicateCounter1.Count();

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

            var duplicates2 = duplicateCounter2.Count();

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

```
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                    }
```

```csharp
                var elapsed1 = sw1.Elapsed;

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

                var sw2 = Stopwatch.StartNew();

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

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

                var elapsed2 = sw2.Elapsed;

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

                Assert.True(elapsed1 > elapsed2);

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

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

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

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

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

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

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

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

                Assert.True(scope1.Links.Count() <= scope2.Links.Count());

                //compressor1.ValidateFrequencies();
            }
        }

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

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

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

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

```csharp
            var strings = new List<string>();

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

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

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

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

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

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

                var sw1 = Stopwatch.StartNew();

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

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

                var elapsed1 = sw1.Elapsed;

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

                var sw2 = Stopwatch.StartNew();

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

                var elapsed2 = sw2.Elapsed;

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

                Assert.True(elapsed1 > elapsed2);

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

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

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

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

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

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

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

                //compressor1.ValidateFrequencies();
            }
        }

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

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

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

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

                var createResults = sequences.CreateAllVariants2(sequence);

                Global.Trash = createResults;

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

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

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

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

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

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

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

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

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

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

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

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

```
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          {
968              sequence[i] = links.Create();
969          }
970
971          var createResults = sequences.CreateAllVariants2(sequence);
972
973          //var reverseResults =
974          ↪   sequences.CreateAllVariants2(sequence.Reverse().ToArray());
975          for (var i = 0; i < 1; i++)
976          {
977              var linksTotalUsages1 = new ulong[links.Count() + 1];
978
979              sequences.CalculateAllUsages(linksTotalUsages1);
980
981              var linksTotalUsages2 = new ulong[links.Count() + 1];
982
983              sequences.CalculateAllUsages2(linksTotalUsages2);
984
985              var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
986              Assert.True(intersection1.Count == linksTotalUsages2.Length);
987          }
988
989          for (var i = 0; i < sequenceLength; i++)
990          {
991              links.Delete(sequence[i]);
992          }
993      }
994  }
995  }
996 }
```

1.123 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs

```
1  using System;
2  using Xunit;
3  using Platform.Memory;
4  using Platform.Data.Doublets.Memory.Split.Generic;
5
6  namespace Platform.Data.Doublets.Tests
7  {
8      public unsafe static class SplitMemoryGenericLinksTests
9      {
10         [Fact]
11         public static void CRUDTest()
12         {
13             Using<byte>(links => links.TestCRUDOperations());
14             Using<ushort>(links => links.TestCRUDOperations());
15             Using<uint>(links => links.TestCRUDOperations());
16             Using<ulong>(links => links.TestCRUDOperations());
17         }
18
```

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

1.124   ./csharp/Platform.Data.Doublets.Tests/TempLinksTestScope.cs

```csharp
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;
```

```
27          Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
28          if (useSequences)
29          {
30              Sequences = new Sequences.Sequences(Links, sequencesOptions);
31          }
32      }
33
34      protected override void Dispose(bool manual, bool wasDisposed)
35      {
36          if (!wasDisposed)
37          {
38              Links.Unsync.DisposeIfPossible();
39              if (_deleteFiles)
40              {
41                  DeleteFiles();
42              }
43          }
44      }
45
46      public void DeleteFiles()
47      {
48          File.Delete(TempFilename);
49          File.Delete(TempTransactionLogFilename);
50      }
51  }
52 }
```

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

```
1  using System.Collections.Generic;
2  using Xunit;
3  using Platform.Ranges;
4  using Platform.Numbers;
5  using Platform.Random;
6  using Platform.Setters;
7  using Platform.Converters;
8
9  namespace Platform.Data.Doublets.Tests
10 {
11     public static class TestExtensions
12     {
13         public static void TestCRUDOperations<T>(this ILinks<T> links)
14         {
15             var constants = links.Constants;
16
17             var equalityComparer = EqualityComparer<T>.Default;
18
19             var zero = default(T);
20             var one = Arithmetic.Increment(zero);
21
22             // Create Link
23             Assert.True(equalityComparer.Equals(links.Count(), zero));
24
25             var setter = new Setter<T>(constants.Null);
26             links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
27
28             Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
29
30             var linkAddress = links.Create();
31
32             var link = new Link<T>(links.GetLink(linkAddress));
33
34             Assert.True(link.Count == 3);
35             Assert.True(equalityComparer.Equals(link.Index, linkAddress));
36             Assert.True(equalityComparer.Equals(link.Source, constants.Null));
37             Assert.True(equalityComparer.Equals(link.Target, constants.Null));
38
39             Assert.True(equalityComparer.Equals(links.Count(), one));
40
41             // Get first link
42             setter = new Setter<T>(constants.Null);
43             links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
44
45             Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
46
47             // Update link to reference itself
48             links.Update(linkAddress, linkAddress, linkAddress);
49
50             link = new Link<T>(links.GetLink(linkAddress));
51
52             Assert.True(equalityComparer.Equals(link.Source, linkAddress));
```

```
53          Assert.True(equalityComparer.Equals(link.Target, linkAddress));
54
55          // Update link to reference null (prepare for delete)
56          var updated = links.Update(linkAddress, constants.Null, constants.Null);
57
58          Assert.True(equalityComparer.Equals(updated, linkAddress));
59
60          link = new Link<T>(links.GetLink(linkAddress));
61
62          Assert.True(equalityComparer.Equals(link.Source, constants.Null));
63          Assert.True(equalityComparer.Equals(link.Target, constants.Null));
64
65          // Delete link
66          links.Delete(linkAddress);
67
68          Assert.True(equalityComparer.Equals(links.Count(), zero));
69
70          setter = new Setter<T>(constants.Null);
71          links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
72
73          Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
74      }
75
76      public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
77      {
78          // Constants
79          var constants = links.Constants;
80          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);
```

```csharp
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;
160             var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
161             var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
162             for (var N = 1; N < maximumOperationsPerCycle; N++)
163             {
164                 var random = new System.Random(N);
165                 var created = 0UL;
166                 var deleted = 0UL;
167                 for (var i = 0; i < N; i++)
168                 {
169                     var linksCount = addressToUInt64Converter.Convert(links.Count());
170                     var createPoint = random.NextBoolean();
171                     if (linksCount > 2 && createPoint)
172                     {
173                         var linksAddressRange = new Range<ulong>(1, linksCount);
174                         TLink source = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA⌋
                            ↪   ddressRange));
175                         TLink target = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA⌋
                            ↪   ddressRange));
                            ↪   //-V3086
176                         var resultLink = links.GetOrCreate(source, target);
177                         if (comparer.Compare(resultLink,
                            ↪   uInt64ToAddressConverter.Convert(linksCount)) > 0)
178                         {
179                             created++;
180                         }
181                     }
182                     else
183                     {
184                         links.Create();
185                         created++;
186                     }
187                 }
188                 Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
189                 for (var i = 0; i < N; i++)
190                 {
191                     TLink link = uInt64ToAddressConverter.Convert((ulong)i + 1UL);
192                     if (links.Exists(link))
193                     {
194                         links.Delete(link);
195                         deleted++;
196                     }
197                 }
198                 Assert.True(addressToUInt64Converter.Convert(links.Count()) == 0L);
199             }
200         }
201     }
202 }
```

1.126   ./csharp/Platform.Data.Doublets.Tests/UInt64LinksTests.cs

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

```csharp
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);
```

```csharp
                    links.Delete(l1);

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

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

            [Fact]
            public static void TransactionAutoRevertedTest()
            {
                // Auto Reverted (Because no commit at transaction)
                using (var scope = new TempLinksTestScope(useLog: true))
                {
                    var links = scope.Links;
                    var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
                    using (var transaction = transactionsLayer.BeginTransaction())
                    {
                        var l1 = links.Create();
                        var l2 = links.Create();

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

                    Assert.Equal(0UL, links.Count());

                    links.Unsync.DisposeIfPossible();

                    var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
                     ↪  cope.TempTransactionLogFilename);
                    Assert.Single(transitions);
                }
            }

            [Fact]
            public static void TransactionUserCodeErrorNoDataSavedTest()
            {
                // User Code Error (Autoreverted), no data saved
                var itself = _constants.Itself;

                TempLinksTestScope lastScope = null;
                try
                {
                    using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                     ↪  useLog: true))
                    {
                        var links = scope.Links;
                        var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
                         ↪  atorBase<ulong>)links.Unsync).Links;
                        using (var transaction = transactionsLayer.BeginTransaction())
                        {
                            var l1 = links.CreateAndUpdate(itself, itself);
                            var l2 = links.CreateAndUpdate(itself, itself);

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

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

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

                            l2 = links.Update(l2, l1);

                            links.Delete(l2);

                            ExceptionThrower();

                            transaction.Commit();
                        }

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

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

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

                lastScope.DeleteFiles();
            }
        }

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

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

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

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

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

                    links.Unsync.DisposeIfPossible();

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

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

                        links.Delete(l2);

                        ExceptionThrower();

                        transaction.Commit();
                    }

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

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

                lastScope.DeleteFiles();
            }
        }

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

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

            // Commit
```

```csharp
228            using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
229            using (var links = new UInt64Links(memoryAdapter))
230            {
231                using (var transaction = memoryAdapter.BeginTransaction())
232                {
233                    var l1 = links.CreateAndUpdate(itself, itself);
234                    var l2 = links.CreateAndUpdate(itself, itself);
235
236                    Global.Trash = links.Update(l2, l2, l1, l2);
237
238                    links.Delete(l1);
239
240                    transaction.Commit();
241                }
242
243                Global.Trash = links.Count();
244            }
245
246            Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪ sactionLogFilename);
247        }
248
249        [Fact]
250        public static void TransactionDamage()
251        {
252            var itself = _constants.Itself;
253
254            var tempDatabaseFilename = Path.GetTempFileName();
255            var tempTransactionLogFilename = Path.GetTempFileName();
256
257            // Commit
258            using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪ 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
```

```csharp
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);
```

```csharp
                        var r2 = links.CheckPathExistance(l2, l2, l2, l2);
                }
            }

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

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

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

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

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

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

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

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

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

            #endregion

            #region Performance

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

            return;

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

                // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
    результат
                // Также это дополнительно помогает в отладке
                // Увеличивает вероятность попадания информации в кэши
                for (var i = 0; i < 10; i++)
```

```
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
```

```csharp
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;
```

```csharp
                    ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
                    ↪ parallel.", Iterations);

                    long counter = 0;

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

                    var sw = Stopwatch.StartNew();

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

                    var elapsedTime = sw.Elapsed;

                    var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                    links.Delete(firstLink);

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

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

                    ulong counter = 0;

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

                    var sw = Stopwatch.StartNew();

                    for (ulong i = 0; i < Iterations; i++)
                    {
                        counter += links.GetTarget(firstLink);
                    }

                    var elapsedTime = sw.Elapsed;

                    var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                    links.Delete(firstLink);

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

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

                    long counter = 0;

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

                    var sw = Stopwatch.StartNew();
```

```csharp
                    Parallel.For(0, Iterations, x =>
                    {
                        Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
                        //Interlocked.Increment(ref counter);
                    });

                    var elapsedTime = sw.Elapsed;

                    var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                    links.Delete(firstLink);

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

        // TODO: Заполнить базу данных перед тестом
        /*
        [Fact]
        public void TestRandomSearchFixed()
        {
            var tempFilename = Path.GetTempFileName();

            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: O(0), was dependent on creation tests")]
        public static void TestRandomSearchAll()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                ulong counter = 0;

                var maxLink = links.Count();

                var iterations = links.Count();

                ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
                    ↪  links.Count());

                var sw = Stopwatch.StartNew();

                for (var i = iterations; i > 0; i--)
```

```csharp
                {
                    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: O(0), was dependent on creation tests")]
        public static void TestEach()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;

                var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);

                ConsoleHelpers.Debug("Testing Each function.");

                var sw = Stopwatch.StartNew();

                links.Each(counter.IncrementAndReturnTrue);

                var elapsedTime = sw.Elapsed;

                var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
                ↪   links per second)",
                    counter, elapsedTime, (long)linksPerSecond);
            }
        }

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

            using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪   DefaultLinksSizeStep))
            {
                ulong counter = 0;

                ConsoleHelpers.Debug("Testing foreach through links.");

                var sw = Stopwatch.StartNew();

                //foreach (var link in links)
                //{
                //    counter++;
                //}

                var elapsedTime = sw.Elapsed;

                var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
    ↪   links per second)", counter, elapsedTime, (long)linksPerSecond);
            }

            File.Delete(tempFilename);
        }
        */

        /*
        [Fact]
        public static void TestParallelForeach()
```

```csharp
            {
                var tempFilename = Path.GetTempFileName();

                using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    DefaultLinksSizeStep))
                {
                    long counter = 0;

                    ConsoleHelpers.Debug("Testing parallel foreach through links.");

                    var sw = Stopwatch.StartNew();

                    //Parallel.ForEach((IEnumerable<ulong>)links, x =>
                    //{
                    //     Interlocked.Increment(ref counter);
                    //});

                    var elapsedTime = sw.Elapsed;

                    var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;

                    ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
    {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
                }

                File.Delete(tempFilename);
            }
            */

            [Fact(Skip = "performance test")]
            public static void Create64BillionLinks()
            {
                using (var scope = new TempLinksTestScope())
                {
                    var links = scope.Links;
                    var linksBeforeTest = links.Count();

                    long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;

                    ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);

                    var elapsedTime = Performance.Measure(() =>
                    {
                        for (long i = 0; i < linksToCreate; i++)
                        {
                            links.Create();
                        }
                    });

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

                    ConsoleHelpers.Debug("Current links count: {0}.", links.Count());

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

            [Fact(Skip = "performance test")]
            public static void Create64BillionLinksInParallel()
            {
                using (var scope = new TempLinksTestScope())
                {
                    var links = scope.Links;
                    var linksBeforeTest = links.Count();

                    var sw = Stopwatch.StartNew();

                    long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;

                    ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);

                    Parallel.For(0, linksToCreate, x => links.Create());

                    var elapsedTime = sw.Elapsed;

                    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,
                   (long)linksPerSecond);
895
896        }
897    }
898
899    [Fact(Skip = "useless: O(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,
                   (long)linksPerSecond);
915
916        }
917    }
918
919    #endregion
920    }
921 }
```

1.127 ./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 }
```

```csharp
using Xunit;
using Platform.Converters;
using Platform.Memory;
using Platform.Reflection;
using Platform.Scopes;
using Platform.Data.Numbers.Raw;
using Platform.Data.Doublets.Incrementers;
using Platform.Data.Doublets.Numbers.Unary;
using Platform.Data.Doublets.PropertyOperators;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Unicode;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.CriterionMatchers;

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

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

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

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

```csharp
                var itself = links.Constants.Itself;

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

                var powerOf2ToUnaryNumberConverter = new
                    PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
                var addressToUnaryNumberConverter = new
                    AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
                var charToUnicodeSymbolConverter = new
                    CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
                    unicodeSymbolMarker);

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

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

                var originalString = "Hello";

                var unicodeSequenceLink =
                    stringToUnicodeSequenceConverter.Convert(originalString);

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

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

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

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

                var resultingString =
                    unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);

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

Index