

LinksPlatform's Platform.Data.Doublets Class Library

1.1 ./csharp/Platform.Data.Doublets/CriterionMatchers/TargetMatcher.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.CriterionMatchers
8 {
9     public class TargetMatcher<TLink> : LinksOperatorBase<TLink>, ICriterionMatcher<TLink>
10    {
11        private static readonly EqualityComparer<TLink> _equalityComparer =
12            ↳ EqualityComparer<TLink>.Default;
13
14        private readonly TLink _targetToMatch;
15
16        [MethodImpl(MethodImplOptions.AggressiveInlining)]
17        public TargetMatcher(ILinks<TLink> links, TLink targetToMatch) : base(links) =>
18            ↳ _targetToMatch = targetToMatch;
19
20        [MethodImpl(MethodImplOptions.AggressiveInlining)]
21        public bool IsMatched(TLink link) => _equalityComparer.Equals(_links.GetTarget(link),
22            ↳ _targetToMatch);
23    }
24 }
```

1.2 ./csharp/Platform.Data.Doublets/Decorators/LinksCascadeUniquenessAndUsagesResolver.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.Decorators
6 {
7     public class LinksCascadeUniquenessAndUsagesResolver<TLink> : LinksUniquenessResolver<TLink>
8    {
9        [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public LinksCascadeUniquenessAndUsagesResolver(ILinks<TLink> links) : base(links) { }
11
12        [MethodImpl(MethodImplOptions.AggressiveInlining)]
13        protected override TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
14            ↳ newLinkAddress)
15        {
16            // Use Facade (the last decorator) to ensure recursion working correctly
17            _facade.MergeUsages(oldLinkAddress, newLinkAddress);
18            return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
19        }
20    }
21 }
```

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

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Decorators
7 {
8     /// <remarks>
9     /// <para>Must be used in conjunction with NonNullContentsLinkDeletionResolver.</para>
10    /// <para>Должен использоваться вместе с NonNullContentsLinkDeletionResolver.</para>
11    /// </remarks>
12    public class LinksCascadeUsagesResolver<TLink> : LinksDecoratorBase<TLink>
13    {
14        [MethodImpl(MethodImplOptions.AggressiveInlining)]
15        public LinksCascadeUsagesResolver(ILinks<TLink> links) : base(links) { }
16
17        [MethodImpl(MethodImplOptions.AggressiveInlining)]
18        public override void Delete(ICollection<TLink> restrictions)
19        {
20            var linkIndex = restrictions[_constants.IndexPart];
21            // Use Facade (the last decorator) to ensure recursion working correctly
22            _facade.DeleteAllUsages(linkIndex);
23            _links.Delete(linkIndex);
24        }
25    }
26 }
```

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

```
1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Decorators
8 {
9     public abstract class LinksDecoratorBase<TLink> : LinksOperatorBase<TLink>, ILinks<TLink>
10    {
11        protected readonly LinksConstants<TLink> _constants;
12
13        public LinksConstants<TLink> Constants
14        {
15            [MethodImpl(MethodImplOptions.AggressiveInlining)]
16            get => _constants;
17        }
18
19        protected ILinks<TLink> _facade;
20
21        public ILinks<TLink> Facade
22        {
23            [MethodImpl(MethodImplOptions.AggressiveInlining)]
24            get => _facade;
25            [MethodImpl(MethodImplOptions.AggressiveInlining)]
26            set
27            {
28                _facade = value;
29                if (_links is LinksDecoratorBase<TLink> decorator)
30                {
31                    decorator.Facade = value;
32                }
33            }
34        }
35
36        [MethodImpl(MethodImplOptions.AggressiveInlining)]
37        protected LinksDecoratorBase(ILinks<TLink> links) : base(links)
38        {
39            _constants = links.Constants;
40            Facade = this;
41        }
42
43        [MethodImpl(MethodImplOptions.AggressiveInlining)]
44        public virtual TLink Count(IList<TLink> restrictions) => _links.Count(restrictions);
45
46        [MethodImpl(MethodImplOptions.AggressiveInlining)]
47        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
48            => _links.Each(handler, restrictions);
49
50        [MethodImpl(MethodImplOptions.AggressiveInlining)]
51        public virtual TLink Create(IList<TLink> restrictions) => _links.Create(restrictions);
52
53        [MethodImpl(MethodImplOptions.AggressiveInlining)]
54        public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
55            _links.Update(restrictions, substitution);
56
57        [MethodImpl(MethodImplOptions.AggressiveInlining)]
58        public virtual void Delete(IList<TLink> restrictions) => _links.Delete(restrictions);
59    }
60 }
```

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

```
1 using System.Runtime.CompilerServices;
2 using Platform.Disposables;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5 #pragma warning disable CA1063 // Implement IDisposable Correctly
6
7 namespace Platform.Data.Doublets.Decorators
8 {
9     public abstract class LinksDisposableDecoratorBase<TLink> : LinksDecoratorBase<TLink>,
10        ILinks<TLink>, System.IDisposable
11    {
12        protected class DisposableWithMultipleCallsAllowed : Disposable
13        {
14            [MethodImpl(MethodImplOptions.AggressiveInlining)]
15            public DisposableWithMultipleCallsAllowed(Disposal disposal) : base(disposal) { }
16
17            protected override bool AllowMultipleDisposeCalls
18            {
19                get { return true; }
20            }
21        }
22    }
23 }
```

```

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
27     => = new DisposableWithMultipleCallsAllowed(Dispose);
28
29 [MethodImpl(MethodImplOptions.AggressiveInlining)]
30 ~LinksDisposableDecoratorBase() => Disposable.Destruct();
31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 public void Dispose() => Disposable.Dispose();
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 protected virtual void Dispose(bool manual, bool wasDisposed)
37 {
38     if (!wasDisposed)
39     {
40         _links.DisposeIfPossible();
41     }
42 }
43 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Decorators
8 {
9     // TODO: Make LinksExternalReferenceValidator. A layer that checks each link to exist or to
10     // be external (hybrid link's raw number).
11     public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
18         {
19             var links = _links;
20             links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
21             return links.Each(handler, restrictions);
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
26         {
27             // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
28             var links = _links;
29             links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
30             links.EnsureInnerReferenceExists(substitution, nameof(substitution));
31             return links.Update(restrictions, substitution);
32         }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public override void Delete(IList<TLink> restrictions)
36         {
37             var link = restrictions[_constants.IndexPart];
38             var links = _links;
39             links.EnsureLinkExists(link, nameof(link));
40             links.Delete(link);
41         }
42     }
43 }

```

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

```

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

```

```

5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Decorators
8 {
9     public class LinksItselfConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
10    {
11        private static readonly EqualityComparer<TLink> _equalityComparer =
12            ↪ EqualityComparer<TLink>.Default;
13
14        [MethodImpl(MethodImplOptions.AggressiveInlining)]
15        public LinksItselfConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
16
17        [MethodImpl(MethodImplOptions.AggressiveInlining)]
18        public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
19        {
20            var constants = _constants;
21            var itselfConstant = constants.Itself;
22            if (!_equalityComparer.Equals(constants.Any, itselfConstant) &&
23                ↪ restrictions.Contains(itselfConstant))
24            {
25                // Itself constant is not supported for Each method right now, skipping execution
26                return constants.Continue;
27            }
28            return _links.Each(handler, restrictions);
29        }
30
31        [MethodImpl(MethodImplOptions.AggressiveInlining)]
32        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
33            ↪ _links.Update(restrictions, _links.ResolveConstantAsSelfReference(_constants.Itself,
34            ↪ restrictions, substitution));
35    }
36 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Decorators
7 {
8     /// <remarks>
9     /// Not practical if newSource and newTarget are too big.
10    /// To be able to use practical version we should allow to create link at any specific
11    ↪ location inside ResizableDirectMemoryLinks.
12    /// This in turn will require to implement not a list of empty links, but a list of ranges
13    ↪ to store it more efficiently.
14    /// </remarks>
15    public class LinksNonExistentDependenciesCreator<TLink> : LinksDecoratorBase<TLink>
16    {
17        [MethodImpl(MethodImplOptions.AggressiveInlining)]
18        public LinksNonExistentDependenciesCreator(ILinks<TLink> links) : base(links) { }
19
20        [MethodImpl(MethodImplOptions.AggressiveInlining)]
21        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
22        {
23            var constants = _constants;
24            var links = _links;
25            links.EnsureCreated(substitution[constants.SourcePart],
26                ↪ substitution[constants.TargetPart]);
27            return links.Update(restrictions, substitution);
28        }
29    }
30 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Decorators
7 {
8     public class LinksNullConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
9    {
10        [MethodImpl(MethodImplOptions.AggressiveInlining)]
11        public LinksNullConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
12
13        [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      public class LinksUniquenessResolver<TLink> : LinksDecoratorBase<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↪ EqualityComparer<TLink>.Default;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public LinksUniquenessResolver(ICollection<TLink> links) : base(links) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public override TLink Update(ICollection<TLink> restrictions, ICollection<TLink> substitution)
18         {
19             var constants = _constants;
20             var links = _links;
21             var newLinkAddress = links.SearchOrDefault(substitution[constants.SourcePart],
22             ↪ substitution[constants.TargetPart]);
23             if (_equalityComparer.Equals(newLinkAddress, default))
24             {
25                 return links.Update(restrictions, substitution);
26             }
27             return ResolveAddressChangeConflict(restrictions[constants.IndexPart],
28             ↪ newLinkAddress);
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
33             ↪ newLinkAddress)
34         {
35             if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
36             ↪ _links.Exists(oldLinkAddress))
37             {
38                 _facade.Delete(oldLinkAddress);
39             }
40             return newLinkAddress;
41         }
42     }
43 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      public class LinksUniquenessValidator<TLink> : LinksDecoratorBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksUniquenessValidator(ICollection<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public override TLink Update(ICollection<TLink> restrictions, ICollection<TLink> substitution)
15         {
16             var links = _links;
17             var constants = _constants;
18             links.EnsureDoesNotExists(substitution[constants.SourcePart],
19             ↪ substitution[constants.TargetPart]);
20             return links.Update(restrictions, substitution);
21         }
22     }
23 }

```

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

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

```
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

```
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
10     ///   with the links storage for links with addresses represented as <see cref="System.UInt64">
11     ///   </para>
12     /// <para>Представляет комбинированный декоратор, реализующий основную логику по
13     ///   взаимодействию с хранилищем связей, для связей с адресами представленными в виде <see
14     ///   cref="System.UInt64"> </para>
15     /// </summary>
16     /// <remarks>
17     /// Возможные оптимизации:
18     /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
19     /// + меньше объём БД
20     /// - меньше производительность
21     /// - больше ограничение на количество связей в БД)
22     /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
```

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using Platform.Collections;
5  using Platform.Collections.Lists;
6  using Platform.Data.Universal;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9

```

```

10 namespace Platform.Data.Doublets.Decorators
11 {
12     /// <remarks>
13     /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
14     /// Now we go with nothing. And nothing is something one, but empty, and cannot be changed
15     /// ↪ by itself. But can cause creation (update from nothing) or deletion (update to nothing).
16     ///
17     /// TODO: Decide to change to IDoubletLinks or not to change. (Better to create
18     /// ↪ DefaultUniLinksBase, that contains logic itself and can be implemented using both
19     /// ↪ IDoubletLinks and ILinks.)
20     /// </remarks>
21     internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
22     {
23         private static readonly EqualityComparer<TLink> _equalityComparer =
24             ↪ EqualityComparer<TLink>.Default;
25
26         public UniLinks(ILinks<TLink> links) : base(links) { }
27
28         private struct Transition
29         {
30             public IList<TLink> Before;
31             public IList<TLink> After;
32
33             public Transition(IList<TLink> before, IList<TLink> after)
34             {
35                 Before = before;
36                 After = after;
37             }
38         }
39
40         //public static readonly TLink NullConstant = Use<LinksConstants<TLink>>.Single.Null;
41         //public static readonly IReadOnlyList<TLink> NullLink = new
42         ↪ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
43         ↪ });
44
45         // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
46         ↪ (Links-Expression)
47         public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
48         ↪ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
49         ↪ substitutedHandler)
50         {
51             //List<Transition> transitions = null;
52             //if (!restriction.IsNullOrEmpty())
53             //if {
54             //    // Есть причина делать проход (чтение)
55             //    if (matchedHandler != null)
56             //    {
57             //        if (!substitution.IsNullOrEmpty())
58             //        {
59             //            // restriction => { 0, 0, 0 } | { 0 } // Create
60             //            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
61             ↪ Create / Update
62             //            // substitution => { 0, 0, 0 } | { 0 } // Delete
63             //            transitions = new List<Transition>();
64             //            if (Equals(substitution[Constants.IndexPart], Constants.Null))
65             //            {
66             //                // If index is Null, that means we always ignore every other
67             ↪ value (they are also Null by definition)
68             //                var matchDecision = matchedHandler(, NullLink);
69             //                if (Equals(matchDecision, Constants.Break))
70             //                    return false;
71             //                if (!Equals(matchDecision, Constants.Skip))
72             //                    transitions.Add(new Transition(matchedLink, newValue));
73             //            }
74             //            else
75             //            {
76             //                Func<T, bool> handler;
77             //                handler = link =>
78             //                {
79             //                    var matchedLink = Memory.GetLinkValue(link);
80             //                    var newValue = Memory.GetLinkValue(link);
81             //                    newValue[Constants.IndexPart] = Constants.Itself;
82             //                    newValue[Constants.SourcePart] =
83             ↪ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
84             ↪ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
85             //                    newValue[Constants.TargetPart] =
86             ↪ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
87             ↪ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];

```



```

73         var matchDecision = matchedHandler(matchedLink, newValue);
74         if (Equals(matchDecision, Constants.Break))
75             return false;
76         if (!Equals(matchDecision, Constants.Skip))
77             transitions.Add(new Transition(matchedLink, newValue));
78         return true;
79     };
80     if (!Memory.Each(handler, restriction))
81         return Constants.Break;
82     }
83     }
84     else
85     {
86         Func<T, bool> handler = link =>
87         {
88             var matchedLink = Memory.GetLinkValue(link);
89             var matchDecision = matchedHandler(matchedLink, matchedLink);
90             return !Equals(matchDecision, Constants.Break);
91         };
92         if (!Memory.Each(handler, restriction))
93             return Constants.Break;
94     }
95     }
96     else
97     {
98         if (substitution != null)
99         {
100             transitions = new List<IList<T>>();
101             Func<T, bool> handler = link =>
102             {
103                 var matchedLink = Memory.GetLinkValue(link);
104                 transitions.Add(matchedLink);
105                 return true;
106             };
107             if (!Memory.Each(handler, restriction))
108                 return Constants.Break;
109         }
110         else
111         {
112             return Constants.Continue;
113         }
114     }
115 }
116 if (substitution != null)
117 {
118     // Есть причина делать замену (запись)
119     if (substitutedHandler != null)
120     {
121     }
122     else
123     {
124     }
125 }
126 return Constants.Continue;
127
128 //if (restriction.IsNullOrEmpty()) // Create
129 //{
130 //    substitution[Constants.IndexPart] = Memory.AllocateLink();
131 //    Memory.SetLinkValue(substitution);
132 //}
133 //else if (substitution.IsNullOrEmpty()) // Delete
134 //{
135 //    Memory.FreeLink(restriction[Constants.IndexPart]);
136 //}
137 //else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
138 //{
139 //    // No need to collect links to list
140 //    // Skip == Continue
141 //    // No need to check substitutedHandler
142 //    if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
143 //        ↪ Constants.Break), restriction))
144 //        return Constants.Break;
145 //}
146 //else // Update
147 //{
148 //    //List<IList<T>> matchedLinks = null;
149 //    if (matchedHandler != null)
150 //    {

```

```

150         //         matchedLinks = new List<ILink<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
174         //                 ↪ substitution handling.
175         //                 var substitutedDecision = substitutedHandler(matchedLink,
176         //                 ↪ newValue);
177         //                 if (Equals(substitutedDecision, Constants.Break))
178         //                     return Constants.Break;
179         //                 if (Equals(substitutedDecision, Constants.Continue))
180         //                 {
181         //                     // Actual update here
182         //                     Memory.SetLinkValue(newValue);
183         //                 }
184         //                 if (Equals(substitutedDecision, Constants.Skip))
185         //                 {
186         //                     // Cancel the update. TODO: decide use separate Cancel
187         //                     ↪ constant or Skip is enough?
188         //                 }
189         //             }
190         //         }
191         //     }
192         // }
193         return _constants.Continue;
194     }
195
196     public TLink Trigger(ILink<TLink> patternOrCondition, Func<ILink<TLink>, TLink>
197     ↪ matchHandler, ILink<TLink> substitution, Func<ILink<TLink>, ILink<TLink>, TLink>
198     ↪ substitutionHandler)
199     {
200         var constants = _constants;
201         if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
202         {
203             return constants.Continue;
204         }
205         else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
206         ↪ Check if it is a correct condition
207         {
208             // Or it only applies to trigger without matchHandler.
209             throw new NotImplementedException();
210         }
211         else if (!substitution.IsNullOrEmpty()) // Creation
212         {
213             var before = Array.Empty<TLink>();
214             // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
215             ↪ (пройти мимо) или пустить (взять)?
216             if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
217             ↪ constants.Break))
218             {
219                 return constants.Break;
220             }
221             var after = (ILink<TLink>)substitution.ToArray();
222             if (_equalityComparer.Equals(after[0], default))
223             {
224                 var newLink = _links.Create();
225                 after[0] = newLink;
226             }
227             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),
244             ↪ constants.Break))
245         {
246             return constants.Break;
247         }
248         var after = Array.Empty<TLink>();
249         _links.Update(linkToDelete, constants.Null, constants.Null);
250         _links.Delete(linkToDelete);
251         if (matchHandler != null)
252         {
253             return substitutionHandler(before, after);
254         }
255         return constants.Continue;
256     }
257     else
258     {
259         throw new NotSupportedException();
260     }
261 }
262 else // Replace / Update
263 {
264     if (patternOrCondition.Count == 1) //-V3125
265     {
266         var linkToUpdate = patternOrCondition[0];
267         var before = _links.GetLink(linkToUpdate);
268         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
269             ↪ constants.Break))
270         {
271             return constants.Break;
272         }
273         var after = (IList<TLink>)substitution.ToArray(); //-V3125
274         if (_equalityComparer.Equals(after[0], default))
275         {
276             after[0] = linkToUpdate;
277         }
278         if (substitution.Count == 1)
279         {
280             if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
281             {
282                 after = _links.GetLink(substitution[0]);
283                 _links.Update(linkToUpdate, constants.Null, constants.Null);
284                 _links.Delete(linkToUpdate);
285             }
286         }
287         else if (substitution.Count == 3)
288         {
289             //Links.Update(after);
290         }
291         else
292         {
293             throw new NotSupportedException();
294         }
295         if (matchHandler != null)
296         {
297             return substitutionHandler(before, after);
298         }
299     }
300 }

```

```

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

}

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

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

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

```

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 =
12             ↳ EqualityComparer<T>.Default;
13
14         public T Source
15         {
16             [MethodImpl(MethodImplOptions.AggressiveInlining)]
17             get;
18             [MethodImpl(MethodImplOptions.AggressiveInlining)]
19             set;
20         }
21
22         public T Target
23         {
24             [MethodImpl(MethodImplOptions.AggressiveInlining)]
25             get;
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             set;
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public Doublet(T source, T target)
32         {
33             Source = source;
34             Target = target;
35         }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public override string ToString() => $"{Source}->{Target}";
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
42             ↳ && _equalityComparer.Equals(Target, other.Target);
43     }
44 }

```

```

40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     public override bool Equals(object obj) => obj is Doublet<T> doublet ?
        ↳ base.Equals(doublet) : false;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     public override int GetHashCode() => (Source, Target).GetHashCode();
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     public static bool operator ==(Doublet<T> left, Doublet<T> right) => left.Equals(right);
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     public static bool operator !=(Doublet<T> left, Doublet<T> right) => !(left == right);
52 }
53 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets
7 {
8     /// <remarks>
9     /// TODO: Может стоит попробовать ref во всех методах (IRefEqualityComparer)
10    /// 2x faster with comparer
11    /// </remarks>
12    public class DoubletComparer<T> : IEqualityComparer<Doublet<T>>
13    {
14        public static readonly DoubletComparer<T> Default = new DoubletComparer<T>();
15
16        [MethodImpl(MethodImplOptions.AggressiveInlining)]
17        public bool Equals(Doublet<T> x, Doublet<T> y) => x.Equals(y);
18
19        [MethodImpl(MethodImplOptions.AggressiveInlining)]
20        public int GetHashCode(Doublet<T> obj) => obj.GetHashCode();
21    }
22 }

```

1.18 ./csharp/Platform.Data.Doublets/ILinks.cs

```

1 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3 using System.Collections.Generic;
4
5 namespace Platform.Data.Doublets
6 {
7     public interface ILinks<TLink> : ILinks<TLink, LinksConstants<TLink>>
8     {
9     }
10 }

```

1.19 ./csharp/Platform.Data.Doublets/ILinksExtensions.cs

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Runtime.CompilerServices;
6 using Platform.Ranges;
7 using Platform.Collections.Arrays;
8 using Platform.Random;
9 using Platform.Setters;
10 using Platform.Converters;
11 using Platform.Numbers;
12 using Platform.Data.Exceptions;
13 using Platform.Data.Doublets.Decorators;
14
15 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
16
17 namespace Platform.Data.Doublets
18 {
19     public static class ILinksExtensions
20     {
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public static void RunRandomCreations<TLink>(this ILinks<TLink> links, ulong
            ↳ amountOfCreations)
23         {
24             var random = RandomHelpers.Default;
25             var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
26             var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;

```

```

27     for (var i = 0UL; i < amountOfCreations; i++)
28     {
29         var linksAddressRange = new Range<ulong>(0,
30             ↳ addressToUInt64Converter.Convert(links.Count()));
31         var source =
32             ↳ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
33         var target =
34             ↳ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
35         links.GetOrCreate(source, target);
36     }
37 }
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 public static void RunRandomSearches<TLink>(this ILinks<TLink> links, ulong
41     ↳ amountOfSearches)
42 {
43     var random = RandomHelpers.Default;
44     var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
45     var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
46     for (var i = 0UL; i < amountOfSearches; i++)
47     {
48         var linksAddressRange = new Range<ulong>(0,
49             ↳ addressToUInt64Converter.Convert(links.Count()));
50         var source =
51             ↳ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
52         var target =
53             ↳ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
54         links.SearchOrDefault(source, target);
55     }
56 }
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, ulong
60     ↳ amountOfDeletions)
61 {
62     var random = RandomHelpers.Default;
63     var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
64     var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
65     var linksCount = addressToUInt64Converter.Convert(links.Count());
66     var min = amountOfDeletions > linksCount ? 0UL : linksCount - amountOfDeletions;
67     for (var i = 0UL; i < amountOfDeletions; i++)
68     {
69         linksCount = addressToUInt64Converter.Convert(links.Count());
70         if (linksCount <= min)
71         {
72             break;
73         }
74         var linksAddressRange = new Range<ulong>(min, linksCount);
75         var link =
76             ↳ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
77         links.Delete(link);
78     }
79 }
80
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 public static void Delete<TLink>(this ILinks<TLink> links, TLink linkToDelete) =>
83     ↳ links.Delete(new LinkAddress<TLink>(linkToDelete));
84
85 /// <remarks>
86 /// TODO: Возможно есть очень простой способ это сделать.
87 /// (Например просто удалить файл, или изменить его размер таким образом,
88 /// чтобы удалился весь контент)
89 /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
90 /// </remarks>
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 public static void DeleteAll<TLink>(this ILinks<TLink> links)
93 {
94     var equalityComparer = EqualityComparer<TLink>.Default;
95     var comparer = Comparer<TLink>.Default;
96     for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
97         ↳ Arithmetic.Decrement(i))
98     {
99         links.Delete(i);
100         if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
101         {
102             i = links.Count();
103         }
104     }
105 }

```

```

94     }
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     public static TLink First<TLink>(this ILinks<TLink> links)
98     {
99         TLink firstLink = default;
100         var equalityComparer = EqualityComparer<TLink>.Default;
101         if (equalityComparer.Equals(links.Count(), default))
102         {
103             throw new InvalidOperationException("В хранилище нет связей.");
104         }
105         links.Each(links.Constants.Any, links.Constants.Any, link =>
106         {
107             firstLink = link[links.Constants.IndexPart];
108             return links.Constants.Break;
109         });
110         if (equalityComparer.Equals(firstLink, default))
111         {
112             throw new InvalidOperationException("В процессе поиска по хранилищу не было
113                 ↳ найдено связей.");
114         }
115         return firstLink;
116     }
117
118     #region Paths
119
120     /// <remarks>
121     /// TODO: Как так? Как то что ниже может быть корректно?
122     /// Скорее всего практически не применимо
123     /// Предполагалось, что можно было конвертировать формируемый в проходе через
124     /// ↳ SequenceWalker
125     /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
126     /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
127     /// </remarks>
128     [MethodImpl(MethodImplOptions.AggressiveInlining)]
129     public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
130         ↳ path)
131     {
132         var current = path[0];
133         //EnsureLinkExists(current, "path");
134         if (!links.Exists(current))
135         {
136             return false;
137         }
138         var equalityComparer = EqualityComparer<TLink>.Default;
139         var constants = links.Constants;
140         for (var i = 1; i < path.Length; i++)
141         {
142             var next = path[i];
143             var values = links.GetLink(current);
144             var source = values[constants.SourcePart];
145             var target = values[constants.TargetPart];
146             if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
147                 ↳ next))
148             {
149                 //throw new InvalidOperationException(string.Format("Невозможно выбрать
150                 ↳ путь, так как и Source и Target совпадают с элементом пути {0}.", next));
151                 return false;
152             }
153             if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
154                 ↳ target))
155             {
156                 //throw new InvalidOperationException(string.Format("Невозможно продолжить
157                 ↳ путь через элемент пути {0}", next));
158                 return false;
159             }
160             current = next;
161         }
162         return true;
163     }
164
165     /// <remarks>
166     /// Может потребовать дополнительного стека для PathElement's при использовании
167     /// ↳ SequenceWalker.
168     /// </remarks>
169     [MethodImpl(MethodImplOptions.AggressiveInlining)]
170     public static TLink GetByKey<TLink>(this ILinks<TLink> links, TLink root, params int[]
171         ↳ path)

```

```

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

```



```

233 /// Возвращает индекс конечной (Target) связи для указанной связи.
234 /// </summary>
235 /// <param name="links">Хранилище связей.</param>
236 /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↳ содержимого.</param>
237 /// <returns>Индекс конечной связи для указанной связи.</returns>
238 [MethodImpl(MethodImplOptions.AggressiveInlining)]
239 public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↳ link[links.Constants.TargetPart];
240
241 /// <summary>
242 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
243 /// </summary>
244 /// <param name="links">Хранилище связей.</param>
245 /// <param name="handler">Обработчик каждой подходящей связи.</param>
246 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
    ↳ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
    ↳ Any - отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
247 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
248 [MethodImpl(MethodImplOptions.AggressiveInlining)]
249 public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
    ↳ handler, params TLink[] restrictions)
250     => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
    ↳ links.Constants.Continue);
251
252 /// <summary>
253 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
254 /// </summary>
255 /// <param name="links">Хранилище связей.</param>
256 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
    ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
257 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
    ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
258 /// <param name="handler">Обработчик каждой подходящей связи.</param>
259 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
260 [MethodImpl(MethodImplOptions.AggressiveInlining)]
261 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
    ↳ Func<TLink, bool> handler)
262 {
263     var constants = links.Constants;
264     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
    ↳ constants.Break, constants.Any, source, target);
265 }
266
267 /// <summary>
268 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
269 /// </summary>
270 /// <param name="links">Хранилище связей.</param>
271 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
    ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
272 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
    ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
273 /// <param name="handler">Обработчик каждой подходящей связи.</param>
274 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
275 [MethodImpl(MethodImplOptions.AggressiveInlining)]
276 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
    ↳ Func<IList<TLink>, TLink> handler) => links.Each(handler, links.Constants.Any,
    ↳ source, target);
277
278 [MethodImpl(MethodImplOptions.AggressiveInlining)]
279 public static IList<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,
286             ↪ links.Constants.Continue);
287         links.Each(filler.AddAndReturnConstant, restrictions);
288         return array;
289     }
290     else
291     {
292         return Array.Empty<IList<TLink>>();
293     }
294 }
295
296 [MethodImpl(MethodImplOptions.AggressiveInlining)]
297 public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
298     ↪ restrictions)
299 {
300     var arraySize = CheckedConverter<TLink,
301         ↪ long>.Default.Convert(links.Count(restrictions));
302     if (arraySize > 0)
303     {
304         var array = new TLink[arraySize];
305         var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
306         links.Each(filler.AddFirstAndReturnConstant, restrictions);
307         return array;
308     }
309     else
310     {
311         return Array.Empty<TLink>();
312     }
313 }
314
315 /// <summary>
316 /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
317 ↪ в хранилище связей.
318 /// </summary>
319 /// <param name="links">Хранилище связей.</param>
320 /// <param name="source">Начало связи.</param>
321 /// <param name="target">Конец связи.</param>
322 /// <returns>Значение, определяющее существует ли связь.</returns>
323 [MethodImpl(MethodImplOptions.AggressiveInlining)]
324 public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
325     ↪ => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
326     ↪ default) > 0;
327
328 #region Ensure
329 // TODO: May be move to EnsureExtensions or make it both there and here
330
331 [MethodImpl(MethodImplOptions.AggressiveInlining)]
332 public static void EnsureLinkExists<TLink>(this ILinks<TLink> links, IList<TLink>
333     ↪ restrictions)
334 {
335     for (var i = 0; i < restrictions.Count; i++)
336     {
337         if (!links.Exists(restrictions[i]))
338         {
339             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
340                 ↪ $"sequence[{i}]");
341         }
342     }
343 }
344
345 [MethodImpl(MethodImplOptions.AggressiveInlining)]
346 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
347     ↪ reference, string argumentName)
348 {
349     if (links.Constants.IsInternalReference(reference) && !links.Exists(reference))
350     {
351         throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
352     }
353 }
354
355 [MethodImpl(MethodImplOptions.AggressiveInlining)]
356 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
357     ↪ IList<TLink> restrictions, string argumentName)
358 {
359     for (int i = 0; i < restrictions.Count; i++)
360     {

```

```

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

```

```

420 {
421     var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
422     var uint64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
423     var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
424         ↪ !links.Exists(x)));
425     if (nonExistentAddresses.Count > 0)
426     {
427         var max = nonExistentAddresses.Max();
428         max = uint64ToAddressConverter.Convert(System.Math.Min(addressToUInt64Converter.
429             ↪ Convert(max),
430             ↪ addressToUInt64Converter.Convert(links.Constants.InternalReferencesRange.Max
431             ↪ imum)));
432         var createdLinks = new List<TLink>();
433         var equalityComparer = EqualityComparer<TLink>.Default;
434         TLink createdLink = creator();
435         while (!equalityComparer.Equals(createdLink, max))
436         {
437             createdLinks.Add(createdLink);
438         }
439         for (var i = 0; i < createdLinks.Count; i++)
440         {
441             if (!nonExistentAddresses.Contains(createdLinks[i]))
442             {
443                 links.Delete(createdLinks[i]);
444             }
445         }
446     }
447 }
448
449 #endregion
450
451 /// <param name="links">Хранилище связей.</param>
452 [MethodImpl(MethodImplOptions.AggressiveInlining)]
453 public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
454 {
455     var constants = links.Constants;
456     var values = links.GetLink(link);
457     TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link,
458         ↪ constants.Any));
459     var equalityComparer = EqualityComparer<TLink>.Default;
460     if (equalityComparer.Equals(values[constants.SourcePart], link))
461     {
462         usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
463     }
464     TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
465         ↪ link));
466     if (equalityComparer.Equals(values[constants.TargetPart], link))
467     {
468         usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
469     }
470     return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
471 }
472
473 /// <param name="links">Хранилище связей.</param>
474 [MethodImpl(MethodImplOptions.AggressiveInlining)]
475 public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
476     ↪ Comparer<TLink>.Default.Compare(links.CountUsages(link), default) > 0;
477
478 /// <param name="links">Хранилище связей.</param>
479 [MethodImpl(MethodImplOptions.AggressiveInlining)]
480 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
481     ↪ TLink target)
482 {
483     var constants = links.Constants;
484     var values = links.GetLink(link);
485     var equalityComparer = EqualityComparer<TLink>.Default;
486     return equalityComparer.Equals(values[constants.SourcePart], source) &&
487         ↪ equalityComparer.Equals(values[constants.TargetPart], target);
488 }
489
490 /// <summary>
491 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
492 /// </summary>
493 /// <param name="links">Хранилище связей.</param>
494 /// <param name="source">Индекс связи, которая является началом для искомой
495     ↪ связи.</param>
496 /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>

```

```

487  /// <returns>Индекс искомой связи с указанными Source (началом) и Target
488  ↪ (концом).</returns>
489  [MethodImpl(MethodImplOptions.AggressiveInlining)]
490  public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
491  ↪ target)
492  {
493      var constants = links.Constants;
494      var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break, default);
495      links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
496      return setter.Result;
497  }
498  /// <param name="links">Хранилище связей.</param>
499  [MethodImpl(MethodImplOptions.AggressiveInlining)]
500  public static TLink Create<TLink>(this ILinks<TLink> links) => links.Create(null);
501  /// <param name="links">Хранилище связей.</param>
502  [MethodImpl(MethodImplOptions.AggressiveInlining)]
503  public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
504  {
505      var link = links.Create();
506      return links.Update(link, link, link);
507  }
508  /// <param name="links">Хранилище связей.</param>
509  [MethodImpl(MethodImplOptions.AggressiveInlining)]
510  public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
511  ↪ target) => links.Update(links.Create(), source, target);
512  /// <summary>
513  /// Обновляет связь с указанными началом (Source) и концом (Target)
514  /// на связь с указанными началом (NewSource) и концом (NewTarget).
515  /// </summary>
516  /// <param name="links">Хранилище связей.</param>
517  /// <param name="link">Индекс обновляемой связи.</param>
518  /// <param name="newSource">Индекс связи, которая является началом связи, на которую
519  ↪ выполняется обновление.</param>
520  /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
521  ↪ выполняется обновление.</param>
522  /// <returns>Индекс обновлённой связи.</returns>
523  [MethodImpl(MethodImplOptions.AggressiveInlining)]
524  public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
525  ↪ TLink newTarget) => links.Update(new LinkAddress<TLink>(link), new Link<TLink>(link,
526  ↪ newSource, newTarget));
527  /// <summary>
528  /// Обновляет связь с указанными началом (Source) и концом (Target)
529  /// на связь с указанными началом (NewSource) и концом (NewTarget).
530  /// </summary>
531  /// <param name="links">Хранилище связей.</param>
532  /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
533  ↪ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
534  ↪ Itself - требование установить ссылку на себя, 1..∞ конкретный адрес другой
535  ↪ связи.</param>
536  /// <returns>Индекс обновлённой связи.</returns>
537  [MethodImpl(MethodImplOptions.AggressiveInlining)]
538  public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
539  {
540      if (restrictions.Length == 2)
541      {
542          return links.MergeAndDelete(restrictions[0], restrictions[1]);
543      }
544      if (restrictions.Length == 4)
545      {
546          return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
547          ↪ restrictions[2], restrictions[3]);
548      }
549      else
550      {
551          return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
552      }
553  }
554  [MethodImpl(MethodImplOptions.AggressiveInlining)]
555  public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
556  ↪ links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
557  {
558      var equalityComparer = EqualityComparer<TLink>.Default;

```

```

553     var constants = links.Constants;
554     var restrictionsIndex = restrictions[constants.IndexPart];
555     var substitutionIndex = substitution[constants.IndexPart];
556     if (equalityComparer.Equals(substitutionIndex, default))
557     {
558         substitutionIndex = restrictionsIndex;
559     }
560     var source = substitution[constants.SourcePart];
561     var target = substitution[constants.TargetPart];
562     source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
563     target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
564     return new Link<TLink>(substitutionIndex, source, target);
565 }
566
567 /// <summary>
568 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
569   ↳ с указанными Source (началом) и Target (концом).
570 /// </summary>
571 /// <param name="links">Хранилище связей.</param>
572 /// <param name="source">Индекс связи, которая является началом на создаваемой
573   ↳ связи.</param>
574 /// <param name="target">Индекс связи, которая является концом для создаваемой
575   ↳ связи.</param>
576 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
577 [MethodImpl(MethodImplOptions.AggressiveInlining)]
578 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
579   ↳ target)
580 {
581     var link = links.SearchOrDefault(source, target);
582     if (EqualityComparer<TLink>.Default.Equals(link, default))
583     {
584         link = links.CreateAndUpdate(source, target);
585     }
586     return link;
587 }
588
589 /// <summary>
590 /// Обновляет связь с указанными началом (Source) и концом (Target)
591   ↳ на связь с указанными началом (NewSource) и концом (NewTarget).
592 /// </summary>
593 /// <param name="links">Хранилище связей.</param>
594 /// <param name="source">Индекс связи, которая является началом обновляемой
595   ↳ связи.</param>
596 /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
597 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
598   ↳ выполняется обновление.</param>
599 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
600   ↳ выполняется обновление.</param>
601 /// <returns>Индекс обновлённой связи.</returns>
602 [MethodImpl(MethodImplOptions.AggressiveInlining)]
603 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
604   ↳ TLink target, TLink newSource, TLink newTarget)
605 {
606     var equalityComparer = EqualityComparer<TLink>.Default;
607     var link = links.SearchOrDefault(source, target);
608     if (equalityComparer.Equals(link, default))
609     {
610         return links.CreateAndUpdate(newSource, newTarget);
611     }
612     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
613   ↳ target))
614     {
615         return link;
616     }
617     return links.Update(link, newSource, newTarget);
618 }
619
620 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
621 /// <param name="links">Хранилище связей.</param>
622 /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
623 /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
624 [MethodImpl(MethodImplOptions.AggressiveInlining)]
625 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
626   ↳ target)
627 {
628     var link = links.SearchOrDefault(source, target);
629     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>
629 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
630 [MethodImpl(MethodImplOptions.AggressiveInlining)]
631 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
632 {
633     for (int i = 0; i < deletedLinks.Count; i++)
634     {
635         links.Delete(deletedLinks[i]);
636     }
637 }
638
639 /// <remarks>Before execution of this method ensure that deleted link is detached (all
640   ↳ values - source and target are reset to null) or it might enter into infinite
641   ↳ recursion.</remarks>
642 [MethodImpl(MethodImplOptions.AggressiveInlining)]
643 public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
644 {
645     var anyConstant = links.Constants.Any;
646     var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
647     links.DeleteByQuery(usagesAsSourceQuery);
648     var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
649     links.DeleteByQuery(usagesAsTargetQuery);
650 }
651
652 [MethodImpl(MethodImplOptions.AggressiveInlining)]
653 public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
654 {
655     var count = CheckedConverter<TLink, long>.Default.Convert(links.Count(query));
656     if (count > 0)
657     {
658         var queryResult = new TLink[count];
659         var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
660   ↳ links.Constants.Continue);
661         links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
662         for (var i = count - 1; i >= 0; i--)
663         {
664             links.Delete(queryResult[i]);
665         }
666     }
667 }
668
669 // TODO: Move to Platform.Data
670 [MethodImpl(MethodImplOptions.AggressiveInlining)]
671 public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
672 {
673     var nullConstant = links.Constants.Null;
674     var equalityComparer = EqualityComparer<TLink>.Default;
675     var link = links.GetLink(linkIndex);
676     for (int i = 1; i < link.Count; i++)
677     {
678         if (!equalityComparer.Equals(link[i], nullConstant))
679         {
680             return false;
681         }
682     }
683     return true;
684 }
685
686 // TODO: Create a universal version of this method in Platform.Data (with using of for
687   ↳ loop)
688 [MethodImpl(MethodImplOptions.AggressiveInlining)]
689 public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
690 {
691     var nullConstant = links.Constants.Null;
692     var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
693     links.Update(updateRequest);
694 }
695
696 // TODO: Create a universal version of this method in Platform.Data (with using of for
697   ↳ loop)

```

```

693 [MethodImpl(MethodImplOptions.AggressiveInlining)]
694 public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
695 {
696     if (!links.AreValuesReset(linkIndex))
697     {
698         links.ResetValues(linkIndex);
699     }
700 }
701
702 /// <summary>
703 /// Merging two usages graphs, all children of old link moved to be children of new link
704   ↳ or deleted.
705 /// </summary>
706 [MethodImpl(MethodImplOptions.AggressiveInlining)]
707 public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
708   ↳ TLink newLinkIndex)
709 {
710     var addressToInt64Converter = CheckedConverter<TLink, long>.Default;
711     var equalityComparer = EqualityComparer<TLink>.Default;
712     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
713     {
714         var constants = links.Constants;
715         var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
716   ↳ constants.Any);
717         var usagesAsSourceCount =
718   ↳ addressToInt64Converter.Convert(links.Count(usagesAsSourceQuery));
719         var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
720   ↳ oldLinkIndex);
721         var usagesAsTargetCount =
722   ↳ addressToInt64Converter.Convert(links.Count(usagesAsTargetQuery));
723         var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
724   ↳ usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
725         if (!isStandalonePoint)
726         {
727             var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
728             if (totalUsages > 0)
729             {
730                 var usages = ArrayPool.Allocate<TLink>(totalUsages);
731                 var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
732   ↳ links.Constants.Continue);
733                 var i = 0L;
734                 if (usagesAsSourceCount > 0)
735                 {
736                     links.Each(usagesFiller.AddFirstAndReturnConstant,
737   ↳ usagesAsSourceQuery);
738                     for (; i < usagesAsSourceCount; i++)
739                     {
740                         var usage = usages[i];
741                         if (!equalityComparer.Equals(usage, oldLinkIndex))
742                         {
743                             links.Update(usage, newLinkIndex, links.GetTarget(usage));
744                         }
745                     }
746                 }
747                 if (usagesAsTargetCount > 0)
748                 {
749                     links.Each(usagesFiller.AddFirstAndReturnConstant,
750   ↳ usagesAsTargetQuery);
751                     for (; i < usages.Length; i++)
752                     {
753                         var usage = usages[i];
754                         if (!equalityComparer.Equals(usage, oldLinkIndex))
755                         {
756                             links.Update(usage, links.GetSource(usage), newLinkIndex);
757                         }
758                     }
759                 }
760                 ArrayPool.Free(usages);
761             }
762         }
763     }
764     return newLinkIndex;
765 }
766
767 /// <summary>
768 /// Replace one link with another (replaced link is deleted, children are updated or
769   ↳ deleted).
770 /// </summary>

```



```

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

```

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

```

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

```

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

```

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

```

```

29     var links = _links;
30     if (_equalityComparer.Equals(frequency, default))
31     {
32         return links.GetOrCreate(_unaryOne, _frequencyMarker);
33     }
34     var incrementedSource =
35         ↪ _unaryNumberIncrementer.Increment(links.GetSource(frequency));
36     return links.GetOrCreate(incrementedSource, _frequencyMarker);
37 }
38 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Incrementers;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Incrementers
8  {
9      public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13
14         private readonly TLink _unaryOne;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
18             ↪ _unaryOne = unaryOne;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public TLink Increment(TLink unaryNumber)
22         {
23             var links = _links;
24             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
25             {
26                 return links.GetOrCreate(_unaryOne, _unaryOne);
27             }
28             var source = links.GetSource(unaryNumber);
29             var target = links.GetTarget(unaryNumber);
30             if (_equalityComparer.Equals(source, target))
31             {
32                 return links.GetOrCreate(unaryNumber, _unaryOne);
33             }
34             else
35             {
36                 return links.GetOrCreate(source, Increment(target));
37             }
38         }
39     }
40 }

```

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

```

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

```

```

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

```

```

100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 public override int GetHashCode() => (Index, Source, Target).GetHashCode();
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
105     && _equalityComparer.Equals(Source, _constants.Null)
106     && _equalityComparer.Equals(Target, _constants.Null);
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 public override bool Equals(object other) => other is Link<TLink> &&
    ↳ Equals((Link<TLink>)other);
110
111 [MethodImpl(MethodImplOptions.AggressiveInlining)]
112 public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
113     && _equalityComparer.Equals(Source, other.Source)
114     && _equalityComparer.Equals(Target, other.Target);
115
116 [MethodImpl(MethodImplOptions.AggressiveInlining)]
117 public static string ToString(TLink index, TLink source, TLink target) => $"{({index}:
    ↳ {source}->{target})}";
118
119 [MethodImpl(MethodImplOptions.AggressiveInlining)]
120 public static string ToString(TLink source, TLink target) => $"{({source}->{target})}";
121
122 [MethodImpl(MethodImplOptions.AggressiveInlining)]
123 public static implicit operator TLink[] (Link<TLink> link) => link.ToArray();
124
125 [MethodImpl(MethodImplOptions.AggressiveInlining)]
126 public static implicit operator Link<TLink>(TLink[] linkArray) => new
    ↳ Link<TLink>(linkArray);
127
128 [MethodImpl(MethodImplOptions.AggressiveInlining)]
129 public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
    ↳ ToString(Source, Target) : ToString(Index, Source, Target);
130
131 #region IList
132
133 public int Count
134 {
135     [MethodImpl(MethodImplOptions.AggressiveInlining)]
136     get => Length;
137 }
138
139 public bool IsReadOnly
140 {
141     [MethodImpl(MethodImplOptions.AggressiveInlining)]
142     get => true;
143 }
144
145 public TLink this[int index]
146 {
147     [MethodImpl(MethodImplOptions.AggressiveInlining)]
148     get
149     {
150         Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
            ↳ nameof(index));
151         if (index == _constants.IndexPart)
152         {
153             return Index;
154         }
155         if (index == _constants.SourcePart)
156         {
157             return Source;
158         }
159         if (index == _constants.TargetPart)
160         {
161             return Target;
162         }
163         throw new NotSupportedException(); // Impossible path due to
            ↳ Ensure.ArgumentInRange
164     }
165     [MethodImpl(MethodImplOptions.AggressiveInlining)]
166     set => throw new NotSupportedException();
167 }
168
169 [MethodImpl(MethodImplOptions.AggressiveInlining)]
170 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
171
172 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

173 public IEnumerator<TLink> GetEnumerator()
174 {
175     yield return Index;
176     yield return Source;
177     yield return Target;
178 }
179
180 [MethodImpl(MethodImplOptions.AggressiveInlining)]
181 public void Add(TLink item) => throw new NotSupportedException();
182
183 [MethodImpl(MethodImplOptions.AggressiveInlining)]
184 public void Clear() => throw new NotSupportedException();
185
186 [MethodImpl(MethodImplOptions.AggressiveInlining)]
187 public bool Contains(TLink item) => IndexOf(item) >= 0;
188
189 [MethodImpl(MethodImplOptions.AggressiveInlining)]
190 public void CopyTo(TLink[] array, int arrayIndex)
191 {
192     Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
193     Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
194         ↪ nameof(arrayIndex));
195     if (arrayIndex + Length > array.Length)
196     {
197         throw new InvalidOperationException();
198     }
199     array[arrayIndex++] = Index;
200     array[arrayIndex++] = Source;
201     array[arrayIndex] = Target;
202 }
203
204 [MethodImpl(MethodImplOptions.AggressiveInlining)]
205 public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
206
207 [MethodImpl(MethodImplOptions.AggressiveInlining)]
208 public int IndexOf(TLink item)
209 {
210     if (_equalityComparer.Equals(Index, item))
211     {
212         return _constants.IndexPart;
213     }
214     if (_equalityComparer.Equals(Source, item))
215     {
216         return _constants.SourcePart;
217     }
218     if (_equalityComparer.Equals(Target, item))
219     {
220         return _constants.TargetPart;
221     }
222     return -1;
223 }
224
225 [MethodImpl(MethodImplOptions.AggressiveInlining)]
226 public void Insert(int index, TLink item) => throw new NotSupportedException();
227
228 [MethodImpl(MethodImplOptions.AggressiveInlining)]
229 public void RemoveAt(int index) => throw new NotSupportedException();
230
231 [MethodImpl(MethodImplOptions.AggressiveInlining)]
232 public static bool operator ==(Link<TLink> left, Link<TLink> right) =>
233     ↪ left.Equals(right);
234
235 [MethodImpl(MethodImplOptions.AggressiveInlining)]
236 public static bool operator !=(Link<TLink> left, Link<TLink> right) => !(left == right);
237
238 #endregion
239 }
240 }

```

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) =>
11         ⇨ Point<TLink>.IsFullPoint(link);
12
13     [MethodImpl(MethodImplOptions.AggressiveInlining)]
14     public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
15         ⇨ Point<TLink>.IsPartialPoint(link);
16 }
17 }

```

1.25 ./csharp/Platform.Data.Doublets/LinksOperatorBase.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 abstract class LinksOperatorBase<TLink>
8     {
9         protected readonly ILinks<TLink> _links;
10
11         public ILinks<TLink> Links
12         {
13             [MethodImpl(MethodImplOptions.AggressiveInlining)]
14             get => _links;
15         }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected LinksOperatorBase(ILinks<TLink> links) => _links = links;
19     }
20 }

```

1.26 ./csharp/Platform.Data.Doublets/Memory/ILinksListMethods.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
6 {
7     public interface ILinksListMethods<TLink>
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         void Detach(TLink freeLink);
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         void AttachAsFirst(TLink link);
14     }
15 }

```

1.27 ./csharp/Platform.Data.Doublets/Memory/ILinksTreeMethods.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.Memory
8 {
9     public interface ILinksTreeMethods<TLink>
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         TLink CountUsages(TLink root);
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         TLink Search(TLink source, TLink target);
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         TLink EachUsage(TLink root, Func<IList<TLink>, TLink> handler);
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         void Detach(ref TLink root, TLink linkIndex);
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         void Attach(ref TLink root, TLink linkIndex);
25     }
26 }

```

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

```

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 =
13             ↳ EqualityComparer<TLink>.Default;
14
15         public static readonly long SizeInBytes = Structure<LinksHeader<TLink>>.Size;
16
17         public TLink AllocatedLinks;
18         public TLink ReservedLinks;
19         public TLink FreeLinks;
20         public TLink FirstFreeLink;
21         public TLink RootAsSource;
22         public TLink RootAsTarget;
23         public TLink LastFreeLink;
24         public TLink Reserved8;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public override bool Equals(object obj) => obj is LinksHeader<TLink> linksHeader ?
28             ↳ Equals(linksHeader) : false;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public bool Equals(LinksHeader<TLink> other)
32             => _equalityComparer.Equals(AllocatedLinks, other.AllocatedLinks)
33             && _equalityComparer.Equals(ReservedLinks, other.ReservedLinks)
34             && _equalityComparer.Equals(FreeLinks, other.FreeLinks)
35             && _equalityComparer.Equals(FirstFreeLink, other.FirstFreeLink)
36             && _equalityComparer.Equals(RootAsSource, other.RootAsSource)
37             && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
38             && _equalityComparer.Equals(LastFreeLink, other.LastFreeLink)
39             && _equalityComparer.Equals(Reserved8, other.Reserved8);
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public override int GetHashCode() => (AllocatedLinks, ReservedLinks, FreeLinks,
43             ↳ FirstFreeLink, RootAsSource, RootAsTarget, LastFreeLink, Reserved8).GetHashCode();
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public static bool operator ==(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
47             ↳ left.Equals(right);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public static bool operator !=(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
51             ↳ !(left == right);
52     }
53 }

```

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

```

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> :
14         ↳ SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
15     {
16         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
17             ↳ UncheckedConverter<TLink, long>.Default;
18
19         protected readonly TLink Break;
20         protected readonly TLink Continue;
21         protected readonly byte* LinksDataParts;
22         protected readonly byte* LinksIndexParts;
23         protected readonly byte* Header;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

24     protected ExternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants,
25         ↳ byte* linksDataParts, byte* linksIndexParts, byte* header)
26     {
27         LinksDataParts = linksDataParts;
28         LinksIndexParts = linksIndexParts;
29         Header = header;
30         Break = constants.Break;
31         Continue = constants.Continue;
32     }
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected abstract TLink GetTreeRoot();
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected abstract TLink GetBasePartValue(TLink link);
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
42         ↳ rootSource, TLink rootTarget);
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
46         ↳ rootSource, TLink rootTarget);
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
50         ↳ AsRef<LinksHeader<TLink>>(Header);
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
54         ↳ AsRef<RawLinkDataPart<TLink>>(LinksDataParts + RawLinkDataPart<TLink>.SizeInBytes *
55         ↳ _addressToInt64Converter.Convert(link));
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
59         ↳ ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
60         ↳ RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link));
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
64     {
65         ref var link = ref GetLinkDataPartReference(linkIndex);
66         return new Link<TLink>(linkIndex, link.Source, link.Target);
67     }
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
71     {
72         ref var firstLink = ref GetLinkDataPartReference(first);
73         ref var secondLink = ref GetLinkDataPartReference(second);
74         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
75             ↳ secondLink.Source, secondLink.Target);
76     }
77
78     [MethodImpl(MethodImplOptions.AggressiveInlining)]
79     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
80     {
81         ref var firstLink = ref GetLinkDataPartReference(first);
82         ref var secondLink = ref GetLinkDataPartReference(second);
83         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
84             ↳ secondLink.Source, secondLink.Target);
85     }
86
87     public TLink this[TLink index]
88     {
89         [MethodImpl(MethodImplOptions.AggressiveInlining)]
90         get
91         {
92             var root = GetTreeRoot();
93             if (GreaterOrEqualThan(index, GetSize(root)))
94             {
95                 return Zero;
96             }
97             while (!EqualToZero(root))
98             {
99                 var left = GetLeftOrDefault(root);
100                 var leftSize = GetSizeOrZero(left);
101                 if (LessThan(index, leftSize))

```



```

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
104                   ↳ broken)
105 }
106
107 /// <summary>
108 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
109   ↳ (концом).
110 /// </summary>
111 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
112 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
113 /// <returns>Индекс искомой связи.</returns>
114 [MethodImpl(MethodImplOptions.AggressiveInlining)]
115 public TLink Search(TLink source, TLink target)
116 {
117     var root = GetTreeRoot();
118     while (!EqualToZero(root))
119     {
120         ref var rootLink = ref GetLinkDataPartReference(root);
121         var rootSource = rootLink.Source;
122         var rootTarget = rootLink.Target;
123         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
124             ↳ node.Key < root.Key
125         {
126             root = GetLeftOrDefault(root);
127         }
128         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
129             ↳ node.Key > root.Key
130         {
131             root = GetRightOrDefault(root);
132         }
133         else // node.Key == root.Key
134         {
135             return root;
136         }
137     }
138     return Zero;
139 }
140
141 // TODO: Return indices range instead of references count
142 [MethodImpl(MethodImplOptions.AggressiveInlining)]
143 public TLink CountUsages(TLink link)
144 {
145     var root = GetTreeRoot();
146     var total = GetSize(root);
147     var totalRightIgnore = Zero;
148     while (!EqualToZero(root))
149     {
150         var @base = GetBasePartValue(root);
151         if (LessOrEqualThan(@base, link))
152         {
153             root = GetRightOrDefault(root);
154         }
155         else
156         {
157             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
158             root = GetLeftOrDefault(root);
159         }
160     }
161     root = GetTreeRoot();
162     var totalLeftIgnore = Zero;
163     while (!EqualToZero(root))
164     {
165         var @base = GetBasePartValue(root);
166         if (GreaterOrEqualThan(@base, link))
167         {
168             root = GetLeftOrDefault(root);
169         }
170     }
171 }

```

```

166     }
167     else
168     {
169         totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
170         root = GetRightOrDefault(root);
171     }
172 }
173 return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
174 }
175
176 [MethodImpl(MethodImplOptions.AggressiveInlining)]
177 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
178     ↳ EachUsageCore(@base, GetTreeRoot(), handler);
179
180 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
181 ↳ low-level MSIL stack.
182 [MethodImpl(MethodImplOptions.AggressiveInlining)]
183 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
184 {
185     var @continue = Continue;
186     if (EqualToZero(link))
187     {
188         return @continue;
189     }
190     var linkBasePart = GetBasePartValue(link);
191     var @break = Break;
192     if (GreaterThan(linkBasePart, @base))
193     {
194         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
195         {
196             return @break;
197         }
198     }
199     else if (LessThan(linkBasePart, @base))
200     {
201         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
202         {
203             return @break;
204         }
205     }
206     else //if (linkBasePart == @base)
207     {
208         if (AreEqual(handler(GetLinkValues(link)), @break))
209         {
210             return @break;
211         }
212         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
213         {
214             return @break;
215         }
216         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
217         {
218             return @break;
219         }
220     }
221     return @continue;
222 }
223
224 [MethodImpl(MethodImplOptions.AggressiveInlining)]
225 protected override void PrintNodeValue(TLink node, StringBuilder sb)
226 {
227     ref var link = ref GetLinkDataPartReference(node);
228     sb.Append(' ');
229     sb.Append(link.Source);
230     sb.Append('-');
231     sb.Append('>');
232     sb.Append(link.Target);
233 }

```

1.30 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSourcesSizeBalancedTreeMethods.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.Split.Generic
6 {

```

```

7 public unsafe class ExternalLinksSourcesSizeBalancedTreeMethods<TLink> :
  ↳ ExternalLinksSizeBalancedTreeMethodsBase<TLink>
8 {
9     [MethodImpl(MethodImplOptions.AggressiveInlining)]
10    public ExternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants,
11    ↳ byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
12    ↳ linksDataParts, linksIndexParts, header) { }
13
14    [MethodImpl(MethodImplOptions.AggressiveInlining)]
15    protected override ref TLink GetLeftReference(TLink node) => ref
16    ↳ GetLinkIndexPartReference(node).LeftAsSource;
17
18    [MethodImpl(MethodImplOptions.AggressiveInlining)]
19    protected override ref TLink GetRightReference(TLink node) => ref
20    ↳ GetLinkIndexPartReference(node).RightAsSource;
21
22    [MethodImpl(MethodImplOptions.AggressiveInlining)]
23    protected override TLink GetLeft(TLink node) =>
24    ↳ GetLinkIndexPartReference(node).LeftAsSource;
25
26    [MethodImpl(MethodImplOptions.AggressiveInlining)]
27    protected override TLink GetRight(TLink node) =>
28    ↳ GetLinkIndexPartReference(node).RightAsSource;
29
30    [MethodImpl(MethodImplOptions.AggressiveInlining)]
31    protected override void SetLeft(TLink node, TLink left) =>
32    ↳ GetLinkIndexPartReference(node).LeftAsSource = left;
33
34    [MethodImpl(MethodImplOptions.AggressiveInlining)]
35    protected override void SetRight(TLink node, TLink right) =>
36    ↳ GetLinkIndexPartReference(node).RightAsSource = right;
37
38    [MethodImpl(MethodImplOptions.AggressiveInlining)]
39    protected override TLink GetSize(TLink node) =>
40    ↳ GetLinkIndexPartReference(node).SizeAsSource;
41
42    [MethodImpl(MethodImplOptions.AggressiveInlining)]
43    protected override void SetSize(TLink node, TLink size) =>
44    ↳ GetLinkIndexPartReference(node).SizeAsSource = size;
45
46    [MethodImpl(MethodImplOptions.AggressiveInlining)]
47    protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
48
49    [MethodImpl(MethodImplOptions.AggressiveInlining)]
50    protected override TLink GetBasePartValue(TLink link) =>
51    ↳ GetLinkDataPartReference(link).Source;
52
53    [MethodImpl(MethodImplOptions.AggressiveInlining)]
54    protected override bool FirstIsToLeftOfSecond(TLink firstSource, TLink firstTarget,
55    ↳ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
56    ↳ AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget);
57
58    [MethodImpl(MethodImplOptions.AggressiveInlining)]
59    protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
60    ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
61    ↳ AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget);
62
63    [MethodImpl(MethodImplOptions.AggressiveInlining)]
64    protected override void ClearNode(TLink node)
65    {
66        ref var link = ref GetLinkIndexPartReference(node);
67        link.LeftAsSource = Zero;
68        link.RightAsSource = Zero;
69        link.SizeAsSource = Zero;
70    }
71 }

```

1.31 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsSizeBalancedTreeMethods.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.Split.Generic
6 {
7     public unsafe class ExternalLinksTargetsSizeBalancedTreeMethods<TLink> :
8     ↳ ExternalLinksSizeBalancedTreeMethodsBase<TLink>
9     {

```

```

9      [MethodImpl(MethodImplOptions.AggressiveInlining)]
10     public ExternalLinksTargetsSizeBalancedTreeMethods(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() => GetHeaderReference().RootAsTarget;

38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected override TLink GetBasePartValue(TLink link) =>
    ↪     GetLinkDataPartReference(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 GetLinkIndexPartReference(node);
52         link.LeftAsTarget = Zero;
53         link.RightAsTarget = Zero;
54         link.SizeAsTarget = Zero;
55     }
56 }
57 }

```

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

```

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

```

```

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);

```

```

158         sb.Append(' ');
159         sb.Append(link.Source);
160         sb.Append('-');
161         sb.Append('>');
162         sb.Append(link.Target);
163     }
164 }
165 }

```

1.33 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesSizeBalancedTreeMethods.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.Split.Generic
6  {
7      public unsafe class InternalLinksSourcesSizeBalancedTreeMethods<TLink> :
8          ↳ InternalLinksSizeBalancedTreeMethodsBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants,
12             ↳ byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
13             ↳ linksDataParts, linksIndexParts, header) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected override ref TLink GetLeftReference(TLink node) => ref
17             ↳ GetLinkIndexPartReference(node).LeftAsSource;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected override ref TLink GetRightReference(TLink node) => ref
21             ↳ GetLinkIndexPartReference(node).RightAsSource;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override TLink GetLeft(TLink node) =>
25             ↳ GetLinkIndexPartReference(node).LeftAsSource;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override TLink GetRight(TLink node) =>
29             ↳ GetLinkIndexPartReference(node).RightAsSource;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetLeft(TLink node, TLink left) =>
33             ↳ GetLinkIndexPartReference(node).LeftAsSource = left;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override void SetRight(TLink node, TLink right) =>
37             ↳ GetLinkIndexPartReference(node).RightAsSource = right;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override TLink GetSize(TLink node) =>
41             ↳ GetLinkIndexPartReference(node).SizeAsSource;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override void SetSize(TLink node, TLink size) =>
45             ↳ GetLinkIndexPartReference(node).SizeAsSource = size;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override TLink GetTreeRoot(TLink link) =>
49             ↳ GetLinkIndexPartReference(link).RootAsSource;
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override TLink GetBasePartValue(TLink link) =>
53             ↳ GetLinkDataPartReference(link).Source;
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override TLink GetKeyPartValue(TLink link) =>
57             ↳ GetLinkDataPartReference(link).Target;
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override void ClearNode(TLink node)
61         {
62             ref var link = ref GetLinkIndexPartReference(node);
63             link.LeftAsSource = Zero;
64             link.RightAsSource = Zero;
65             link.SizeAsSource = Zero;
66         }
67
68         public override TLink Search(TLink source, TLink target) =>
69             ↳ SearchCore(GetTreeRoot(source), target);

```

```

55     }
56 }

```

1.34 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsSizeBalancedTreeMethods.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.Split.Generic
6  {
7      public unsafe class InternalLinksTargetsSizeBalancedTreeMethods<TLink> :
8          ↳ InternalLinksSizeBalancedTreeMethodsBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public InternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants,
12             ↳ byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
13             ↳ linksDataParts, linksIndexParts, header) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected override ref TLink GetLeftReference(TLink node) => ref
17             ↳ GetLinkIndexPartReference(node).LeftAsTarget;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected override ref TLink GetRightReference(TLink node) => ref
21             ↳ GetLinkIndexPartReference(node).RightAsTarget;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override TLink GetLeft(TLink node) =>
25             ↳ GetLinkIndexPartReference(node).LeftAsTarget;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override TLink GetRight(TLink node) =>
29             ↳ GetLinkIndexPartReference(node).RightAsTarget;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetLeft(TLink node, TLink left) =>
33             ↳ GetLinkIndexPartReference(node).LeftAsTarget = left;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override void SetRight(TLink node, TLink right) =>
37             ↳ GetLinkIndexPartReference(node).RightAsTarget = right;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override TLink GetSize(TLink node) =>
41             ↳ GetLinkIndexPartReference(node).SizeAsTarget;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override void SetSize(TLink node, TLink size) =>
45             ↳ GetLinkIndexPartReference(node).SizeAsTarget = size;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override TLink GetTreeRoot(TLink link) =>
49             ↳ GetLinkIndexPartReference(link).RootAsTarget;
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override TLink GetBasePartValue(TLink link) =>
53             ↳ GetLinkDataPartReference(link).Target;
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override TLink GetKeyPartValue(TLink link) =>
57             ↳ GetLinkDataPartReference(link).Source;
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override void ClearNode(TLink node)
61         {
62             ref var link = ref GetLinkIndexPartReference(node);
63             link.LeftAsTarget = Zero;
64             link.RightAsTarget = Zero;
65             link.SizeAsTarget = Zero;
66         }
67
68         public override TLink Search(TLink source, TLink target) =>
69             ↳ SearchCore(GetTreeRoot(target), source);
70     }
71 }

```

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

```

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.Split.Generic
10 {
11     public unsafe class SplitMemoryLinks<TLink> : SplitMemoryLinksBase<TLink>
12     {
13         private readonly Func<ILinksTreeMethods<TLink>> _createInternalSourceTreeMethods;
14         private readonly Func<ILinksTreeMethods<TLink>> _createExternalSourceTreeMethods;
15         private readonly Func<ILinksTreeMethods<TLink>> _createInternalTargetTreeMethods;
16         private readonly Func<ILinksTreeMethods<TLink>> _createExternalTargetTreeMethods;
17         private byte* _header;
18         private byte* _linksDataParts;
19         private byte* _linksIndexParts;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
23             ↪ indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
27             ↪ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
28             ↪ memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
32             ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
33             ↪ base(dataMemory, indexMemory, memoryReservationStep, constants)
34         {
35             _createInternalSourceTreeMethods = () => new
36                 ↪ InternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
37                 ↪ _linksIndexParts, _header);
38             _createExternalSourceTreeMethods = () => new
39                 ↪ ExternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
40                 ↪ _linksIndexParts, _header);
41             _createInternalTargetTreeMethods = () => new
42                 ↪ InternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
43                 ↪ _linksIndexParts, _header);
44             _createExternalTargetTreeMethods = () => new
45                 ↪ ExternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
46                 ↪ _linksIndexParts, _header);
47             Init(dataMemory, indexMemory, memoryReservationStep);
48         }
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override void SetPointers(IResizableDirectMemory dataMemory,
52             ↪ IResizableDirectMemory indexMemory)
53         {
54             _linksDataParts = (byte*)dataMemory.Pointer;
55             _linksIndexParts = (byte*)indexMemory.Pointer;
56             _header = _linksIndexParts;
57             InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
58             ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
59             InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
60             ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
61             UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_linksDataParts, _header);
62         }
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected override void ResetPointers()
66         {
67             base.ResetPointers();
68             _linksDataParts = null;
69             _linksIndexParts = null;
70             _header = null;
71         }
72
73         [MethodImpl(MethodImplOptions.AggressiveInlining)]
74         protected override ref LinksHeader<TLink> GetHeaderReference() => ref
75             ↪ AsRef<LinksHeader<TLink>>(_header);
76
77         [MethodImpl(MethodImplOptions.AggressiveInlining)]
78         protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
79             ↪ => ref AsRef<RawLinkDataPart<TLink>>(_linksDataParts + LinkDataPartSizeInBytes *
80             ↪ ConvertToInt64(linkIndex));
81     }
82 }

```

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5  using Platform.Singletons;
6  using Platform.Converters;
7  using Platform.Numbers;
8  using Platform.Memory;
9  using Platform.Data.Exceptions;
10
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Data.Doublets.Memory.Split.Generic
14 {
15     public abstract class SplitMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16     {
17         private static readonly EqualityComparer<TLink> _equalityComparer =
18             ↪ EqualityComparer<TLink>.Default;
19         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
20         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
21             ↪ UncheckedConverter<TLink, long>.Default;
22         private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
23             ↪ UncheckedConverter<long, TLink>.Default;
24
25         private static readonly TLink _zero = default;
26         private static readonly TLink _one = Arithmetic.Increment(_zero);
27
28         /// <summary>Возвращает размер одной связи в байтах.</summary>
29         /// <remarks>
30         /// Используется только во вне класса, не рекомендуется использовать внутри.
31         /// Так как во вне не обязательно будет доступен unsafe C#.
32         /// </remarks>
33         public static readonly long LinkDataPartSizeInBytes = RawLinkDataPart<TLink>.SizeInBytes;
34
35         public static readonly long LinkIndexPartSizeInBytes =
36             ↪ RawLinkIndexPart<TLink>.SizeInBytes;
37
38         public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
39
40         public static readonly long DefaultLinksSizeStep = 1 * 1024 * 1024;
41
42         protected readonly IResizableDirectMemory _dataMemory;
43         protected readonly IResizableDirectMemory _indexMemory;
44         protected readonly long _dataMemoryReservationStepInBytes;
45         protected readonly long _indexMemoryReservationStepInBytes;
46
47         protected ILinksTreeMethods<TLink> InternalSourcesTreeMethods;
48         protected ILinksTreeMethods<TLink> ExternalSourcesTreeMethods;
49         protected ILinksTreeMethods<TLink> InternalTargetsTreeMethods;
50         protected ILinksTreeMethods<TLink> ExternalTargetsTreeMethods;
51         // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
52         ↪ нужно использовать не список а дерево, так как так можно быстрее проверить на
53         ↪ наличие связи внутри
54         protected ILinksListMethods<TLink> UnusedLinksListMethods;
55
56         /// <summary>
57         /// Возвращает общее число связей находящихся в хранилище.
58         /// </summary>
59         protected virtual TLink Total
60         {
61             [MethodImpl(MethodImplOptions.AggressiveInlining)]
62             get
63             {
64                 ref var header = ref GetHeaderReference();
65                 return Subtract(header.AllocatedLinks, header.FreeLinks);
66             }
67         }
68
69         public virtual LinksConstants<TLink> Constants
70         {
71             [MethodImpl(MethodImplOptions.AggressiveInlining)]
72             get;
73         }
74     }
75 }

```

```

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

```

```

136     }
137     else
138     {
139         return Add(InternalSourcesTreeMethods.CountUsages(value),
140             ↪ InternalTargetsTreeMethods.CountUsages(value));
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) ||
154         ↪ AreEqual(storedLinkValue.Target, value))
155     {
156         return GetOne();
157     }
158     return GetZero();
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 &&
174                 ↪ externalReferencesRange.Value.Contains(target))
175             {
176                 return ExternalTargetsTreeMethods.CountUsages(target);
177             }
178             else
179             {
180                 return InternalTargetsTreeMethods.CountUsages(target);
181             }
182         }
183         else if (AreEqual(target, any))
184         {
185             if (externalReferencesRange.HasValue &&
186                 ↪ externalReferencesRange.Value.Contains(source))
187             {
188                 return ExternalSourcesTreeMethods.CountUsages(source);
189             }
190             else
191             {
192                 return InternalSourcesTreeMethods.CountUsages(source);
193             }
194         }
195         else //if(source != Any && target != Any)
196         {
197             // ЭКВИВАЛЕНТ Exists(source, target) => Count(Any, source, target) > 0
198             TLink link;
199             if (externalReferencesRange.HasValue)
200             {
201                 if (externalReferencesRange.Value.Contains(source) &&
202                     ↪ externalReferencesRange.Value.Contains(target))
203                 {
204                     link = ExternalSourcesTreeMethods.Search(source, target);
205                 }
206                 else if (externalReferencesRange.Value.Contains(source))
207                 {
208                     link = InternalTargetsTreeMethods.Search(source, target);
209                 }
210                 else if (externalReferencesRange.Value.Contains(target))
211                 {
212                     link = InternalTargetsTreeMethods.Search(source, target);
213                 }
214             }
215             else
216             {
217                 link = InternalTargetsTreeMethods.Search(source, target);
218             }
219         }
220     }
221 }

```

```

209         link = InternalSourcesTreeMethods.Search(source, target);
210     }
211     else
212     {
213         if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
214             ↪ InternalTargetsTreeMethods.CountUsages(target)))
215         {
216             link = InternalTargetsTreeMethods.Search(source, target);
217         }
218         else
219         {
220             link = InternalSourcesTreeMethods.Search(source, target);
221         }
222     }
223     else
224     {
225         if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
226             ↪ InternalTargetsTreeMethods.CountUsages(target)))
227         {
228             link = InternalTargetsTreeMethods.Search(source, target);
229         }
230         else
231         {
232             link = InternalSourcesTreeMethods.Search(source, target);
233         }
234     }
235     return AreEqual(link, constants.Null) ? GetZero() : GetOne();
236 }
237 else
238 {
239     if (!Exists(index))
240     {
241         return GetZero();
242     }
243     if (AreEqual(source, any) && AreEqual(target, any))
244     {
245         return GetOne();
246     }
247     ref var storedLinkValue = ref GetLinkDataPartReference(index);
248     if (!AreEqual(source, any) && !AreEqual(target, any))
249     {
250         if (AreEqual(storedLinkValue.Source, source) &&
251             ↪ AreEqual(storedLinkValue.Target, target))
252         {
253             return GetOne();
254         }
255         return GetZero();
256     }
257     var value = default(TLink);
258     if (AreEqual(source, any))
259     {
260         value = target;
261     }
262     if (AreEqual(target, any))
263     {
264         value = source;
265     }
266     if (AreEqual(storedLinkValue.Source, value) ||
267         ↪ AreEqual(storedLinkValue.Target, value))
268     {
269         return GetOne();
270     }
271     return GetZero();
272 }
273 }
274
275 throw new NotSupportedException("Другие размеры и способы ограничений не
276     ↪ поддерживаются.");
277
278 [MethodImpl(MethodImplOptions.AggressiveInlining)]
279 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
280 {
281     var constants = Constants;
282     var @break = constants.Break;
283     if (restrictions.Count == 0)
284     {

```

```

282     for (var link = GetOne(); LessOrEqualThan(link,
283         ↪ GetHeaderReference().AllocatedLinks); link = Increment(link))
284     {
285         if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
286         {
287             return @break;
288         }
289     }
290     return @break;
291 }
292 var @continue = constants.Continue;
293 var any = constants.Any;
294 var index = restrictions[constants.IndexPart];
295 if (restrictions.Count == 1)
296 {
297     if (AreEqual(index, any))
298     {
299         return Each(handler, Array.Empty<TLink>());
300     }
301     if (!Exists(index))
302     {
303         return @continue;
304     }
305     return handler(GetLinkStruct(index));
306 }
307 if (restrictions.Count == 2)
308 {
309     var value = restrictions[1];
310     if (AreEqual(index, any))
311     {
312         if (AreEqual(value, any))
313         {
314             return Each(handler, Array.Empty<TLink>());
315         }
316         if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
317         {
318             return @break;
319         }
320         return Each(handler, new Link<TLink>(index, any, value));
321     }
322     else
323     {
324         if (!Exists(index))
325         {
326             return @continue;
327         }
328         if (AreEqual(value, any))
329         {
330             return handler(GetLinkStruct(index));
331         }
332         ref var storedLinkValue = ref GetLinkDataPartReference(index);
333         if (AreEqual(storedLinkValue.Source, value) ||
334             AreEqual(storedLinkValue.Target, value))
335         {
336             return handler(GetLinkStruct(index));
337         }
338         return @continue;
339     }
340 }
341 if (restrictions.Count == 3)
342 {
343     var externalReferencesRange = constants.ExternalReferencesRange;
344     var source = restrictions[constants.SourcePart];
345     var target = restrictions[constants.TargetPart];
346     if (AreEqual(index, any))
347     {
348         if (AreEqual(source, any) && AreEqual(target, any))
349         {
350             return Each(handler, Array.Empty<TLink>());
351         }
352         else if (AreEqual(source, any))
353         {
354             if (externalReferencesRange.HasValue &&
355                 ↪ externalReferencesRange.Value.Contains(target))
356             {
357                 return ExternalTargetsTreeMethods.EachUsage(target, handler);
358             }
359             else

```

```

358         {
359             return InternalTargetsTreeMethods.EachUsage(target, handler);
360         }
361     }
362     else if (AreEqual(target, any))
363     {
364         if (externalReferencesRange.HasValue &&
365             ↪ externalReferencesRange.Value.Contains(source))
366         {
367             return ExternalSourcesTreeMethods.EachUsage(source, handler);
368         }
369         else
370         {
371             return InternalSourcesTreeMethods.EachUsage(source, handler);
372         }
373     }
374     else //if(source != Any && target != Any)
375     {
376         TLink link;
377         if (externalReferencesRange.HasValue)
378         {
379             if (externalReferencesRange.Value.Contains(source) &&
380                 ↪ externalReferencesRange.Value.Contains(target))
381             {
382                 link = ExternalSourcesTreeMethods.Search(source, target);
383             }
384             else if (externalReferencesRange.Value.Contains(source))
385             {
386                 link = InternalTargetsTreeMethods.Search(source, target);
387             }
388             else if (externalReferencesRange.Value.Contains(target))
389             {
390                 link = InternalSourcesTreeMethods.Search(source, target);
391             }
392             else
393             {
394                 if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
395                     ↪ InternalTargetsTreeMethods.CountUsages(target)))
396                 {
397                     link = InternalTargetsTreeMethods.Search(source, target);
398                 }
399                 else
400                 {
401                     link = InternalSourcesTreeMethods.Search(source, target);
402                 }
403             }
404         }
405         else
406         {
407             if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
408                 ↪ InternalTargetsTreeMethods.CountUsages(target)))
409             {
410                 link = InternalTargetsTreeMethods.Search(source, target);
411             }
412             else
413             {
414                 link = InternalSourcesTreeMethods.Search(source, target);
415             }
416         }
417         return AreEqual(link, constants.Null) ? @continue :
418             ↪ handler(GetLinkStruct(link));
419     }
420 }
421 else
422 {
423     if (!Exists(index))
424     {
425         return @continue;
426     }
427     if (AreEqual(source, any) && AreEqual(target, any))
428     {
429         return handler(GetLinkStruct(index));
430     }
431     ref var storedLinkValue = ref GetLinkDataPartReference(index);
432     if (!AreEqual(source, any) && !AreEqual(target, any))
433     {
434         if (AreEqual(storedLinkValue.Source, source) &&
435             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
458 /// ↳ в другом месте (но не в менеджере памяти, а в логике Links)
459 /// </remarks>
460 [MethodImpl(MethodImplOptions.AggressiveInlining)]
461 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
462 {
463     var constants = Constants;
464     var @null = constants.Null;
465     var externalReferencesRange = constants.ExternalReferencesRange;
466     var linkIndex = restrictions[constants.IndexPart];
467     ref var link = ref GetLinkDataPartReference(linkIndex);
468     var source = link.Source;
469     var target = link.Target;
470     ref var header = ref GetHeaderReference();
471     ref var rootAsSource = ref header.RootAsSource;
472     ref var rootAsTarget = ref header.RootAsTarget;
473     // Будет корректно работать только в том случае, если пространство выделенной связи
474     // ↳ предварительно заполнено нулями
475     if (!AreEqual(source, @null))
476     {
477         if (externalReferencesRange.HasValue &&
478             ↳ externalReferencesRange.Value.Contains(source))
479         {
480             ExternalSourcesTreeMethods.Detach(ref rootAsSource, linkIndex);
481         }
482         else
483         {
484             InternalSourcesTreeMethods.Detach(ref
485                 ↳ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
486         }
487     }
488     if (!AreEqual(target, @null))
489     {
490         if (externalReferencesRange.HasValue &&
491             ↳ externalReferencesRange.Value.Contains(target))
492         {
493             ExternalTargetsTreeMethods.Detach(ref rootAsTarget, linkIndex);
494         }
495         else
496         {
497             InternalTargetsTreeMethods.Detach(ref
498                 ↳ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
499         }
500     }
501     source = link.Source = substitution[constants.SourcePart];
502     target = link.Target = substitution[constants.TargetPart];
503     if (!AreEqual(source, @null))
504     {
505         if (externalReferencesRange.HasValue &&
506             ↳ externalReferencesRange.Value.Contains(source))
507         {

```



```

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

```

```

573         while (GreaterThan(header.AllocatedLinks, GetZero()) &&
574             ↪ IsUnusedLink(header.AllocatedLinks))
575         {
576             UnusedLinksListMethods.Detach(header.AllocatedLinks);
577             header.AllocatedLinks = Decrement(header.AllocatedLinks);
578             _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
579             _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
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
592     /// ↪ адрес реально поменялся
593     ///
594     /// Указатель this.links может быть в том же месте,
595     /// так как 0-я связь не используется и имеет такой же размер как Header,
596     /// поэтому header размещается в том же месте, что и 0-я связь
597     /// </remarks>
598     [MethodImpl(MethodImplOptions.AggressiveInlining)]
599     protected abstract void SetPointers(IResizableDirectMemory dataMemory,
600     ↪ IResizableDirectMemory indexMemory);
601
602     [MethodImpl(MethodImplOptions.AggressiveInlining)]
603     protected virtual void ResetPointers()
604     {
605         InternalSourcesTreeMethods = null;
606         ExternalSourcesTreeMethods = null;
607         InternalTargetsTreeMethods = null;
608         ExternalTargetsTreeMethods = null;
609         UnusedLinksListMethods = null;
610     }
611
612     [MethodImpl(MethodImplOptions.AggressiveInlining)]
613     protected abstract ref LinksHeader<TLink> GetHeaderReference();
614
615     [MethodImpl(MethodImplOptions.AggressiveInlining)]
616     protected abstract ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex);
617
618     [MethodImpl(MethodImplOptions.AggressiveInlining)]
619     protected abstract ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
620     ↪ linkIndex);
621
622     [MethodImpl(MethodImplOptions.AggressiveInlining)]
623     protected virtual bool Exists(TLink link)
624     => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
625     && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
626     && !IsUnusedLink(link);
627
628     [MethodImpl(MethodImplOptions.AggressiveInlining)]
629     protected virtual bool IsUnusedLink(TLink linkIndex)
630     {
631         if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
632         ↪ is not needed
633         {
634             // TODO: Reduce access to memory in different location (should be enough to use
635             ↪ just linkIndexPart)
636             ref var linkDataPart = ref GetLinkDataPartReference(linkIndex);
637             ref var linkIndexPart = ref GetLinkIndexPartReference(linkIndex);
638             return AreEqual(linkIndexPart.SizeAsSource, default) &&
639             ↪ !AreEqual(linkDataPart.Source, default);
640         }
641         else
642         {
643             return true;
644         }
645     }
646
647     [MethodImpl(MethodImplOptions.AggressiveInlining)]
648     protected virtual TLink GetOne() => _one;
649
650     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Methods.Lists;
3  using Platform.Converters;
4  using static System.Runtime.CompilerServices.Unsafe;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Memory.Split.Generic
9  {
10     public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
        ↳ ILinksListMethods<TLink>

```

```

11 {
12     private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
13         ↪ UncheckedConverter<TLink, long>.Default;
14
15     private readonly byte* _links;
16     private readonly byte* _header;
17
18     [MethodImpl(MethodImplOptions.AggressiveInlining)]
19     public UnusedLinksListMethods(byte* links, byte* header)
20     {
21         _links = links;
22         _header = header;
23     }
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
27         ↪ AsRef<LinksHeader<TLink>>(_header);
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
31         ↪ AsRef<RawLinkDataPart<TLink>>(_links + RawLinkDataPart<TLink>.SizeInBytes *
32         ↪ _addressToInt64Converter.Convert(link));
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override TLink GetPrevious(TLink element) =>
42         ↪ GetLinkDataPartReference(element).Source;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override TLink GetNext(TLink element) =>
46         ↪ GetLinkDataPartReference(element).Target;
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected override TLink GetSize() => GetHeaderReference().FreeLinks;
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
53         ↪ element;
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
57         ↪ element;
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override void SetPrevious(TLink element, TLink previous) =>
61         ↪ GetLinkDataPartReference(element).Source = previous;
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override void SetNext(TLink element, TLink next) =>
65         ↪ GetLinkDataPartReference(element).Target = next;
66
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
69 }
70 }

```

1.38 ./csharp/Platform.Data.Doublets/Memory/Split/RawLinkDataPart.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 RawLinkDataPart<TLink> : IEquatable<RawLinkDataPart<TLink>>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↪ EqualityComparer<TLink>.Default;
14
15         public static readonly long SizeInBytes = Structure<RawLinkDataPart<TLink>>.Size;
16
17         public TLink Source;
18         public TLink Target;
19     }
20 }

```

```

18     [MethodImpl(MethodImplOptions.AggressiveInlining)]
19     public override bool Equals(object obj) => obj is RawLinkDataPart<TLink> link ?
20         ↳ 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>
32         ↳ right) => left.Equals(right);
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     public static bool operator !=(RawLinkDataPart<TLink> left, RawLinkDataPart<TLink>
36         ↳ right) => !(left == right);
37 }
38 }

```

1.39 ./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 =
13             ↳ EqualityComparer<TLink>.Default;
14
15         public static readonly long SizeInBytes = Structure<RawLinkIndexPart<TLink>>.Size;
16
17         public TLink RootAsSource;
18         public TLink LeftAsSource;
19         public TLink RightAsSource;
20         public TLink SizeAsSource;
21         public TLink RootAsTarget;
22         public TLink LeftAsTarget;
23         public TLink RightAsTarget;
24         public TLink SizeAsTarget;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public override bool Equals(object obj) => obj is RawLinkIndexPart<TLink> link ?
28             ↳ Equals(link) : false;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public bool Equals(RawLinkIndexPart<TLink> other)
32             => _equalityComparer.Equals(RootAsSource, other.RootAsSource)
33             && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
34             && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
35             && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
36             && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
37             && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
38             && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
39             && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public override int GetHashCode() => (RootAsSource, LeftAsSource, RightAsSource,
43             ↳ SizeAsSource, RootAsTarget, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public static bool operator ==(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
47             ↳ right) => left.Equals(right);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public static bool operator !=(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
51             ↳ right) => !(left == right);
52     }
53 }

```

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

```
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 Platform.Numbers;
8 using static System.Runtime.CompilerServices.Unsafe;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.Memory.United.Generic
13 {
14     public unsafe abstract class LinksAvlBalancedTreeMethodsBase<TLink> :
15         ↳ SizedAndThreadedAVLBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
16     {
17         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
18             ↳ UncheckedConverter<TLink, long>.Default;
19         private static readonly UncheckedConverter<TLink, int> _addressToInt32Converter =
20             ↳ UncheckedConverter<TLink, int>.Default;
21         private static readonly UncheckedConverter<bool, TLink> _boolToAddressConverter =
22             ↳ UncheckedConverter<bool, TLink>.Default;
23         private static readonly UncheckedConverter<TLink, bool> _addressToBoolConverter =
24             ↳ UncheckedConverter<TLink, bool>.Default;
25         private static readonly UncheckedConverter<int, TLink> _int32ToAddressConverter =
26             ↳ UncheckedConverter<int, TLink>.Default;
27
28         protected readonly TLink Break;
29         protected readonly TLink Continue;
30         protected readonly byte* Links;
31         protected readonly byte* Header;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
35             ↳ byte* header)
36         {
37             Links = links;
38             Header = header;
39             Break = constants.Break;
40             Continue = constants.Continue;
41         }
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected abstract TLink GetTreeRoot();
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected abstract TLink GetBasePartValue(TLink link);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
51             ↳ rootSource, TLink rootTarget);
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
55             ↳ rootSource, TLink rootTarget);
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
59             ↳ AsRef<LinksHeader<TLink>>(Header);
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
63             ↳ AsRef<RawLink<TLink>>(Links + RawLink<TLink>.SizeInBytes *
64             ↳ _addressToInt64Converter.Convert(link));
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
68         {
69             ref var link = ref GetLinkReference(linkIndex);
70             return new Link<TLink>(linkIndex, link.Source, link.Target);
71         }
72
73         [MethodImpl(MethodImplOptions.AggressiveInlining)]
74         protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
75         {
76             ref var firstLink = ref GetLinkReference(first);
77             ref var secondLink = ref GetLinkReference(second);
78             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
79                 ↳ secondLink.Source, secondLink.Target);
80         }
81     }
82 }
```

```

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,
75             ↪ secondLink.Source, secondLink.Target);
76     }
77
78     [MethodImpl(MethodImplOptions.AggressiveInlining)]
79     protected virtual TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5,
80         ↪ -5);
81
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     protected virtual void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
84         ↪ Bit<TLink>.PartialWrite(storedValue, size, 5, -5);
85
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     protected virtual bool GetLeftIsChildValue(TLink value)
88     {
89         unchecked
90         {
91             return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 4, 1));
92             //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
93         }
94     }
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     protected virtual void SetLeftIsChildValue(ref TLink storedValue, bool value)
98     {
99         unchecked
100         {
101             var previousValue = storedValue;
102             var modified = Bit<TLink>.PartialWrite(previousValue,
103                 ↪ _boolToAddressConverter.Convert(value), 4, 1);
104             storedValue = modified;
105         }
106     }
107
108     [MethodImpl(MethodImplOptions.AggressiveInlining)]
109     protected virtual bool GetRightIsChildValue(TLink value)
110     {
111         unchecked
112         {
113             return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 3, 1));
114             //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
115         }
116     }
117
118     [MethodImpl(MethodImplOptions.AggressiveInlining)]
119     protected virtual void SetRightIsChildValue(ref TLink storedValue, bool value)
120     {
121         unchecked
122         {
123             var previousValue = storedValue;
124             var modified = Bit<TLink>.PartialWrite(previousValue,
125                 ↪ _boolToAddressConverter.Convert(value), 3, 1);
126             storedValue = modified;
127         }
128     }
129
130     [MethodImpl(MethodImplOptions.AggressiveInlining)]
131     protected bool IsChild(TLink parent, TLink possibleChild)
132     {
133         var parentSize = GetSize(parent);
134         var childSize = GetSizeOrZero(possibleChild);
135         return GreaterThanZero(childSize) && LessOrEqualThan(childSize, parentSize);
136     }
137
138     [MethodImpl(MethodImplOptions.AggressiveInlining)]
139     protected virtual sbyte GetBalanceValue(TLink storedValue)
140     {
141         unchecked
142         {
143             var value = _addressToInt32Converter.Convert(Bit<TLink>.PartialRead(storedValue,
144                 ↪ 0, 3));
145         }
146     }

```

```

139         value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
140         ↪ end of sbyte
141         return (sbyte)value;
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 |
150         ↪ value & 3);
151         var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
152         storedValue = modified;
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
182         ↪ broken)
183     }
184 }
185 /// <summary>
186 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
187 ↪ (концом).
188 /// </summary>
189 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
190 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
191 /// <returns>Индекс искомой связи.</returns>
192 [MethodImpl(MethodImplOptions.AggressiveInlining)]
193 public TLink Search(TLink source, TLink target)
194 {
195     var root = GetTreeRoot();
196     while (!EqualToZero(root))
197     {
198         ref var rootLink = ref GetLinkReference(root);
199         var rootSource = rootLink.Source;
200         var rootTarget = rootLink.Target;
201         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
202         ↪ node.Key < root.Key
203         {
204             root = GetLeftOrDefault(root);
205         }
206         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
207         ↪ node.Key > root.Key
208         {
209             root = GetRightOrDefault(root);
210         }
211         else // node.Key == root.Key
212         {
213             return root;
214         }
215     }
216 }

```



```

211     }
212 }
213 return Zero;
214 }
215
216 // TODO: Return indices range instead of references count
217 [MethodImpl(MethodImplOptions.AggressiveInlining)]
218 public TLink CountUsages(TLink link)
219 {
220     var root = GetTreeRoot();
221     var total = GetSize(root);
222     var totalRightIgnore = Zero;
223     while (!EqualToZero(root))
224     {
225         var @base = GetBasePartValue(root);
226         if (LessOrEqualThan(@base, link))
227         {
228             root = GetRightOrDefault(root);
229         }
230         else
231         {
232             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
233             root = GetLeftOrDefault(root);
234         }
235     }
236     root = GetTreeRoot();
237     var totalLeftIgnore = Zero;
238     while (!EqualToZero(root))
239     {
240         var @base = GetBasePartValue(root);
241         if (GreaterOrEqualThan(@base, link))
242         {
243             root = GetLeftOrDefault(root);
244         }
245         else
246         {
247             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
248             root = GetRightOrDefault(root);
249         }
250     }
251     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
252 }
253
254 [MethodImpl(MethodImplOptions.AggressiveInlining)]
255 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
256 {
257     var root = GetTreeRoot();
258     if (EqualToZero(root))
259     {
260         return Continue;
261     }
262     TLink first = Zero, current = root;
263     while (!EqualToZero(current))
264     {
265         var @base = GetBasePartValue(current);
266         if (GreaterOrEqualThan(@base, link))
267         {
268             if (AreEqual(@base, link))
269             {
270                 first = current;
271             }
272             current = GetLeftOrDefault(current);
273         }
274         else
275         {
276             current = GetRightOrDefault(current);
277         }
278     }
279     if (!EqualToZero(first))
280     {
281         current = first;
282         while (true)
283         {
284             if (AreEqual(handler(GetLinkValues(current)), Break))
285             {
286                 return Break;
287             }
288             current = GetNext(current);
289         }
290     }

```

```

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 }

```

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

```

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> :
14         ↳ SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
15     {
16         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
17             ↳ UncheckedConverter<TLink, long>.Default;
18
19         protected readonly TLink Break;
20         protected readonly TLink Continue;
21         protected readonly byte* Links;
22         protected readonly byte* Header;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
26             ↳ byte* header)
27         {
28             Links = links;
29             Header = header;
30             Break = constants.Break;
31             Continue = constants.Continue;
32         }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected abstract TLink GetTreeRoot();
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected abstract TLink GetBasePartValue(TLink link);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
42             ↳ rootSource, TLink rootTarget);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
46             ↳ rootSource, TLink rootTarget);
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
50             ↳ AsRef<LinksHeader<TLink>>(Header);
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
54             ↳ AsRef<RawLink<TLink>>(Links + RawLink<TLink>.SizeInBytes *
55             ↳ _addressToInt64Converter.Convert(link));
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

50     protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
51     {
52         ref var link = ref GetLinkReference(linkIndex);
53         return new Link<TLink>(linkIndex, link.Source, link.Target);
54     }
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected override bool FirstIsToLeftOfSecond(TLink first, TLink second)
58     {
59         ref var firstLink = ref GetLinkReference(first);
60         ref var secondLink = ref GetLinkReference(second);
61         return FirstIsToLeftOfSecond(firstLink.Source, firstLink.Target,
62             ↪ secondLink.Source, secondLink.Target);
63     }
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
67     {
68         ref var firstLink = ref GetLinkReference(first);
69         ref var secondLink = ref GetLinkReference(second);
70         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
71             ↪ secondLink.Source, secondLink.Target);
72     }
73
74     public TLink this[TLink index]
75     {
76         [MethodImpl(MethodImplOptions.AggressiveInlining)]
77         get
78         {
79             var root = GetTreeRoot();
80             if (GreaterOrEqualThan(index, GetSize(root)))
81             {
82                 return Zero;
83             }
84             while (!EqualToZero(root))
85             {
86                 var left = GetLeftOrDefault(root);
87                 var leftSize = GetSizeOrZero(left);
88                 if (LessThan(index, leftSize))
89                 {
90                     root = left;
91                     continue;
92                 }
93                 if (AreEqual(index, leftSize))
94                 {
95                     return root;
96                 }
97                 root = GetRightOrDefault(root);
98                 index = Subtract(index, Increment(leftSize));
99             }
100             return Zero; // TODO: Impossible situation exception (only if tree structure
101                 ↪ broken)
102         }
103     }
104
105     /// <summary>
106     /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
107     /// ↪ (концом).
108     /// </summary>
109     /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
110     /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
111     /// <returns>Индекс искомой связи.</returns>
112     [MethodImpl(MethodImplOptions.AggressiveInlining)]
113     public TLink Search(TLink source, TLink target)
114     {
115         var root = GetTreeRoot();
116         while (!EqualToZero(root))
117         {
118             ref var rootLink = ref GetLinkReference(root);
119             var rootSource = rootLink.Source;
120             var rootTarget = rootLink.Target;
121             if (FirstIsToLeftOfSecond(source, target, rootSource, rootTarget)) //
122                 ↪ node.Key < root.Key
123             {
124                 root = GetLeftOrDefault(root);
125             }
126             else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
127                 ↪ node.Key > root.Key

```

```

122         {
123             root = GetRightOrDefault(root);
124         }
125         else // node.Key == root.Key
126         {
127             return root;
128         }
129     }
130     return Zero;
131 }
132
133 // TODO: Return indices range instead of references count
134 [MethodImpl(MethodImplOptions.AggressiveInlining)]
135 public TLink CountUsages(TLink link)
136 {
137     var root = GetTreeRoot();
138     var total = GetSize(root);
139     var totalRightIgnore = Zero;
140     while (!EqualToZero(root))
141     {
142         var @base = GetBasePartValue(root);
143         if (LessOrEqualThan(@base, link))
144         {
145             root = GetRightOrDefault(root);
146         }
147         else
148         {
149             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
150             root = GetLeftOrDefault(root);
151         }
152     }
153     root = GetTreeRoot();
154     var totalLeftIgnore = Zero;
155     while (!EqualToZero(root))
156     {
157         var @base = GetBasePartValue(root);
158         if (GreaterOrEqualThan(@base, link))
159         {
160             root = GetLeftOrDefault(root);
161         }
162         else
163         {
164             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
165             root = GetRightOrDefault(root);
166         }
167     }
168     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
169 }
170
171 [MethodImpl(MethodImplOptions.AggressiveInlining)]
172 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
173     ↳ EachUsageCore(@base, GetTreeRoot(), handler);
174
175 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
176 ↳ low-level MSIL stack.
177 [MethodImpl(MethodImplOptions.AggressiveInlining)]
178 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
179 {
180     var @continue = Continue;
181     if (EqualToZero(link))
182     {
183         return @continue;
184     }
185     var linkBasePart = GetBasePartValue(link);
186     var @break = Break;
187     if (GreaterThan(linkBasePart, @base))
188     {
189         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
190         {
191             return @break;
192         }
193     }
194     else if (LessThan(linkBasePart, @base))
195     {
196         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
197         {
198             return @break;
199         }
200     }
201 }

```

```

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

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

```

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

```

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> :
8         ↳ LinksAvlBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksSourcesAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
12             ↳ byte* header) : base(constants, links, header) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref TLink GetLeftReference(TLink node) => ref
16             ↳ GetLinkReference(node).LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref TLink GetRightReference(TLink node) => ref
20             ↳ GetLinkReference(node).RightAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(TLink node, TLink left) =>
30             ↳ GetLinkReference(node).LeftAsSource = left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(TLink node, TLink right) =>
34             ↳ GetLinkReference(node).RightAsSource = right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override TLink GetSize(TLink node) =>
38             ↳ GetSizeValue(GetLinkReference(node).SizeAsSource);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
42             ↳ GetLinkReference(node).SizeAsSource, size);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override bool GetLeftIsChild(TLink node) =>
46             ↳ GetLeftIsChildValue(GetLinkReference(node).SizeAsSource);
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected override bool GetRightIsChild(TLink node) =>
50             ↳ GetRightIsChildValue(GetLinkReference(node).SizeAsSource);
51     }
52 }

```

```

39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 protected override void SetLeftIsChild(TLink node, bool value) =>
    ↳ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected override bool GetRightIsChild(TLink node) =>
    ↳ GetRightIsChildValue(GetLinkReference(node).SizeAsSource);
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 protected override void SetRightIsChild(TLink node, bool value) =>
    ↳ SetRightIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 protected override sbyte GetBalance(TLink node) =>
    ↳ GetBalanceValue(GetLinkReference(node).SizeAsSource);
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
    ↳ GetLinkReference(node).SizeAsSource, value);
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
    ↳ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
    ↳ AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget);
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
    ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
    ↳ AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget);
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 protected override void ClearNode(TLink node)
68 {
69     ref var link = ref GetLinkReference(node);
70     link.LeftAsSource = Zero;
71     link.RightAsSource = Zero;
72     link.SizeAsSource = Zero;
73 }
74 }
75 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Memory.United.Generic
6 {
7     public unsafe class LinksSourcesSizeBalancedTreeMethods<TLink> :
8         ↳ LinksSizeBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
12             ↳ byte* header) : base(constants, links, header) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref TLink GetLeftReference(TLink node) => ref
16             ↳ GetLinkReference(node).LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref TLink GetRightReference(TLink node) => ref
20             ↳ GetLinkReference(node).RightAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(TLink node, TLink left) =>
30             ↳ GetLinkReference(node).LeftAsSource = left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(TLink node, TLink right) =>
34             ↳ GetLinkReference(node).RightAsSource = right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
38             ↳ GetLinkReference(node).SizeAsSource, value);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override void SetLeftIsChild(TLink node, bool value) =>
42             ↳ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override void SetRightIsChild(TLink node, bool value) =>
46             ↳ SetRightIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected override sbyte GetBalance(TLink node) => GetBalanceValue(GetLinkReference(
50             ↳ node).SizeAsSource);
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override bool GetLeftIsChild(TLink node) => GetLeftIsChildValue(GetLinkReference(
54             ↳ node).SizeAsSource);
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override bool GetRightIsChild(TLink node) => GetRightIsChildValue(GetLinkReference(
58             ↳ node).SizeAsSource);
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
62             ↳ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
63             ↳ AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget);
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
67             ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
68             ↳ AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget);
69
70         [MethodImpl(MethodImplOptions.AggressiveInlining)]
71         protected override void ClearNode(TLink node)
72         {
73             ref var link = ref GetLinkReference(node);
74             link.LeftAsSource = Zero;
75             link.RightAsSource = Zero;
76             link.SizeAsSource = Zero;
77         }
78     }
79 }

```

```

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

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Memory.United.Generic
6 {
7     public unsafe class LinksTargetsAvlBalancedTreeMethods<TLink> :
        ↳ LinksAvlBalancedTreeMethodsBase<TLink>
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public LinksTargetsAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
        ↳ byte* header) : base(constants, links, header) { }
11
12        [MethodImpl(MethodImplOptions.AggressiveInlining)]
13        protected override ref TLink GetLeftReference(TLink node) => ref
        ↳ GetLinkReference(node).LeftAsTarget;
14
15        [MethodImpl(MethodImplOptions.AggressiveInlining)]
16        protected override ref TLink GetRightReference(TLink node) => ref
        ↳ GetLinkReference(node).RightAsTarget;
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
20
21        [MethodImpl(MethodImplOptions.AggressiveInlining)]
22        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
23
24        [MethodImpl(MethodImplOptions.AggressiveInlining)]
25        protected override void SetLeft(TLink node, TLink left) =>
        ↳ GetLinkReference(node).LeftAsTarget = left;
26
27        [MethodImpl(MethodImplOptions.AggressiveInlining)]
28        protected override void SetRight(TLink node, TLink right) =>
        ↳ GetLinkReference(node).RightAsTarget = right;
29
30        [MethodImpl(MethodImplOptions.AggressiveInlining)]
31        protected override TLink GetSize(TLink node) =>
        ↳ GetSizeValue(GetLinkReference(node).SizeAsTarget);
32
33        [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

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

```

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

```



```

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) =>
26     ↳ GetLinkReference(node).LeftAsTarget = left;
27
28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 protected override void SetRight(TLink node, TLink right) =>
30     ↳ GetLinkReference(node).RightAsTarget = right;
31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsTarget;
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 protected override void SetSize(TLink node, TLink size) =>
37     ↳ GetLinkReference(node).SizeAsTarget = size;
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
47     ↳ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
48     ↳ AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource);
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
52     ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
53     ↳ AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource);
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 protected override void ClearNode(TLink node)
57 {
58     ref var link = ref GetLinkReference(node);
59     link.LeftAsTarget = Zero;
60     link.RightAsTarget = Zero;
61     link.SizeAsTarget = Zero;
62 }
63 }
64 }

```

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

```

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
23         ↳ минимальным шагом расширения базы данных.
24         /// </summary>
25         /// <param name="address">Полный путь к файлу базы данных.</param>
26         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
27         ↳ байтах.</param>
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
30     ↳ FileMappedResizableDirectMemory(address, memoryReservationStep),
31     ↳ memoryReservationStep) { }

```

```

29 [MethodImpl(MethodImplOptions.AggressiveInlining)]
30 public UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
    ↳ DefaultLinksSizeStep) { }
31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep) :
    ↳ this(memory, memoryReservationStep, Default<LinksConstants<TLink>>.Instance, true) {
    ↳ }
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep,
    ↳ LinksConstants<TLink> constants, bool useAvlBasedIndex) : base(memory,
    ↳ memoryReservationStep, constants)
37 {
38     if (useAvlBasedIndex)
39     {
40         _createSourceTreeMethods = () => new
41             ↳ LinksSourcesAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
42         _createTargetTreeMethods = () => new
43             ↳ LinksTargetsAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
44     }
45     else
46     {
47         _createSourceTreeMethods = () => new
48             ↳ LinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
49         _createTargetTreeMethods = () => new
50             ↳ LinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
51     }
52     Init(memory, memoryReservationStep);
53 }
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 protected override void SetPointers(IResizableDirectMemory memory)
57 {
58     _links = (byte*)memory.Pointer;
59     _header = _links;
60     SourcesTreeMethods = _createSourceTreeMethods();
61     TargetsTreeMethods = _createTargetTreeMethods();
62     UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_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<TLink> GetHeaderReference() => ref
75     ↳ AsRef<LinksHeader<TLink>>(_header);
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected override ref RawLink<TLink> GetLinkReference(TLink linkIndex) => ref
79     ↳ AsRef<RawLink<TLink>>(_links + LinkSizeInBytes * ConvertToInt64(linkIndex));
80 }
81 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Disposables;
5 using Platform.Singletons;
6 using Platform.Converters;
7 using Platform.Numbers;
8 using Platform.Memory;
9 using Platform.Data.Exceptions;
10
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Data.Doublets.Memory.United.Generic
14 {
15     public abstract class UnitedMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16     {
17         private static readonly EqualityComparer<TLink> _equalityComparer =
18             ↳ EqualityComparer<TLink>.Default;
19         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
20     }
21 }

```

```

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

```

```

90 public virtual TLink Count(IList<TLink> restrictions)
91 {
92     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
93     if (restrictions.Count == 0)
94     {
95         return Total;
96     }
97     var constants = Constants;
98     var any = constants.Any;
99     var index = restrictions[constants.IndexPart];
100     if (restrictions.Count == 1)
101     {
102         if (AreEqual(index, any))
103         {
104             return Total;
105         }
106         return Exists(index) ? GetOne() : GetZero();
107     }
108     if (restrictions.Count == 2)
109     {
110         var value = restrictions[1];
111         if (AreEqual(index, any))
112         {
113             if (AreEqual(value, any))
114             {
115                 return Total; // Any - как отсутствие ограничения
116             }
117             return Add(SourcesTreeMethods.CountUsages(value),
118                 ↪ TargetsTreeMethods.CountUsages(value));
119         }
120         else
121         {
122             if (!Exists(index))
123             {
124                 return GetZero();
125             }
126             if (AreEqual(value, any))
127             {
128                 return GetOne();
129             }
130             ref var storedLinkValue = ref GetLinkReference(index);
131             if (AreEqual(storedLinkValue.Source, value) ||
132                 ↪ AreEqual(storedLinkValue.Target, value))
133             {
134                 return GetOne();
135             }
136             return GetZero();
137         }
138     }
139     if (restrictions.Count == 3)
140     {
141         var source = restrictions[constants.SourcePart];
142         var target = restrictions[constants.TargetPart];
143         if (AreEqual(index, any))
144         {
145             if (AreEqual(source, any) && AreEqual(target, any))
146             {
147                 return Total;
148             }
149             else if (AreEqual(source, any))
150             {
151                 return TargetsTreeMethods.CountUsages(target);
152             }
153             else if (AreEqual(target, any))
154             {
155                 return SourcesTreeMethods.CountUsages(source);
156             }
157             else //if(source != Any && target != Any)
158             {
159                 // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
160                 var link = SourcesTreeMethods.Search(source, target);
161                 return AreEqual(link, constants.Null) ? GetZero() : GetOne();
162             }
163         }
164         else
165         {
166             if (!Exists(index))
167             {

```

```

166         return GetZero();
167     }
168     if (AreEqual(source, any) && AreEqual(target, any))
169     {
170         return GetOne();
171     }
172     ref var storedLinkValue = ref GetLinkReference(index);
173     if (!AreEqual(source, any) && !AreEqual(target, any))
174     {
175         if (AreEqual(storedLinkValue.Source, source) &&
176             ↪ AreEqual(storedLinkValue.Target, target))
177         {
178             return GetOne();
179         }
180         return GetZero();
181     }
182     var value = default(TLink);
183     if (AreEqual(source, any))
184     {
185         value = target;
186     }
187     if (AreEqual(target, any))
188     {
189         value = source;
190     }
191     if (AreEqual(storedLinkValue.Source, value) ||
192         ↪ AreEqual(storedLinkValue.Target, value))
193     {
194         return GetOne();
195     }
196     return GetZero();
197 }
198 }
199
200 [MethodImpl(MethodImplOptions.AggressiveInlining)]
201 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
202 {
203     var constants = Constants;
204     var @break = constants.Break;
205     if (restrictions.Count == 0)
206     {
207         for (var link = GetOne(); LessOrEqualThan(link,
208             ↪ GetHeaderReference().AllocatedLinks); link = Increment(link))
209         {
210             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
211             {
212                 return @break;
213             }
214         }
215         return @break;
216     }
217     var @continue = constants.Continue;
218     var any = constants.Any;
219     var index = restrictions[constants.IndexPart];
220     if (restrictions.Count == 1)
221     {
222         if (AreEqual(index, any))
223         {
224             return Each(handler, Array.Empty<TLink>());
225         }
226         if (!Exists(index))
227         {
228             return @continue;
229         }
230         return handler(GetLinkStruct(index));
231     }
232     if (restrictions.Count == 2)
233     {
234         var value = restrictions[1];
235         if (AreEqual(index, any))
236         {
237             if (AreEqual(value, any))
238             {
239                 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 :
287                 ↪ handler(GetLinkStruct(link));
288         }
289     }
290     else
291     {
292         if (!Exists(index))
293         {
294             return @continue;
295         }
296         if (AreEqual(source, any) && AreEqual(target, any))
297         {
298             return handler(GetLinkStruct(index));
299         }
300         ref var storedLinkValue = ref GetLinkReference(index);
301         if (!AreEqual(source, any) && !AreEqual(target, any))
302         {
303             if (AreEqual(storedLinkValue.Source, source) &&
304                 AreEqual(storedLinkValue.Target, target))
305             {
306                 return handler(GetLinkStruct(index));
307             }
308             return @continue;
309         }
310         var value = default(TLink);
311         if (AreEqual(source, any))
312         {
313             value = target;
314         }
315         if (AreEqual(target, any))
316         {
317             value = source;

```

```

317     }
318     if (AreEqual(storedLinkValue.Source, value) ||
319         AreEqual(storedLinkValue.Target, value))
320     {
321         return handler(GetLinkStruct(index));
322     }
323     return @continue;
324 }
325 }
326 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
327 }
328
329 /// <remarks>
330 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↳ в другом месте (но не в менеджере памяти, а в логике Links)
331 /// </remarks>
332 [MethodImpl(MethodImplOptions.AggressiveInlining)]
333 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
334 {
335     var constants = Constants;
336     var @null = constants.Null;
337     var linkIndex = restrictions[constants.IndexPart];
338     ref var link = ref GetLinkReference(linkIndex);
339     ref var header = ref GetHeaderReference();
340     ref var firstAsSource = ref header.RootAsSource;
341     ref var firstAsTarget = ref header.RootAsTarget;
342     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↳ предварительно заполнено нулями
343     if (!AreEqual(link.Source, @null))
344     {
345         SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
346     }
347     if (!AreEqual(link.Target, @null))
348     {
349         TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
350     }
351     link.Source = substitution[constants.SourcePart];
352     link.Target = substitution[constants.TargetPart];
353     if (!AreEqual(link.Source, @null))
354     {
355         SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
356     }
357     if (!AreEqual(link.Target, @null))
358     {
359         TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
360     }
361     return linkIndex;
362 }
363
364 /// <remarks>
365 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
    ↳ пространство
366 /// </remarks>
367 [MethodImpl(MethodImplOptions.AggressiveInlining)]
368 public virtual TLink Create(IList<TLink> restrictions)
369 {
370     ref var header = ref GetHeaderReference();
371     var freeLink = header.FirstFreeLink;
372     if (!AreEqual(freeLink, Constants.Null))
373     {
374         UnusedLinksListMethods.Detach(freeLink);
375     }
376     else
377     {
378         var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
379         if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
380         {
381             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
382         }
383         if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
384         {
385             _memory.ReservedCapacity += _memory.ReservationStep;
386             SetPointers(_memory);
387             header = ref GetHeaderReference();
388             header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
    ↳ LinkSizeInBytes);
389         }
390     }

```

```

390         header.AllocatedLinks = Increment(header.AllocatedLinks);
391         _memory.UsedCapacity += LinkSizeInBytes;
392         freeLink = header.AllocatedLinks;
393     }
394     return freeLink;
395 }
396
397 [MethodImpl(MethodImplOptions.AggressiveInlining)]
398 public virtual void Delete(ICollection<TLink> restrictions)
399 {
400     ref var header = ref GetHeaderReference();
401     var link = restrictions[Constants.IndexPart];
402     if (LessThan(link, header.AllocatedLinks))
403     {
404         UnusedLinksListMethods.AttachAsFirst(link);
405     }
406     else if (AreEqual(link, header.AllocatedLinks))
407     {
408         header.AllocatedLinks = Decrement(header.AllocatedLinks);
409         _memory.UsedCapacity -= LinkSizeInBytes;
410         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
411         //   ↳ пока не дойдём до первой существующей связи
412         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
413         while (GreaterThan(header.AllocatedLinks, GetZero()) &&
414             ↳ IsUnusedLink(header.AllocatedLinks))
415         {
416             UnusedLinksListMethods.Detach(header.AllocatedLinks);
417             header.AllocatedLinks = Decrement(header.AllocatedLinks);
418             _memory.UsedCapacity -= LinkSizeInBytes;
419         }
420     }
421 }
422
423 [MethodImpl(MethodImplOptions.AggressiveInlining)]
424 public ICollection<TLink> GetLinkStruct(TLink linkIndex)
425 {
426     ref var link = ref GetLinkReference(linkIndex);
427     return new Link<TLink>(linkIndex, link.Source, link.Target);
428 }
429
430 /// <remarks>
431 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
432 ///   ↳ адрес реально поменялся
433 ///
434 /// Указатель this.links может быть в том же месте,
435 /// так как 0-я связь не используется и имеет такой же размер как Header,
436 /// поэтому header размещается в том же месте, что и 0-я связь
437 /// </remarks>
438 [MethodImpl(MethodImplOptions.AggressiveInlining)]
439 protected abstract void SetPointers(IResizableDirectMemory memory);
440
441 [MethodImpl(MethodImplOptions.AggressiveInlining)]
442 protected virtual void ResetPointers()
443 {
444     SourcesTreeMethods = null;
445     TargetsTreeMethods = null;
446     UnusedLinksListMethods = null;
447 }
448
449 [MethodImpl(MethodImplOptions.AggressiveInlining)]
450 protected abstract ref LinksHeader<TLink> GetHeaderReference();
451
452 [MethodImpl(MethodImplOptions.AggressiveInlining)]
453 protected abstract ref RawLink<TLink> GetLinkReference(TLink linkIndex);
454
455 [MethodImpl(MethodImplOptions.AggressiveInlining)]
456 protected virtual bool Exists(TLink link)
457 => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
458     && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
459     && !IsUnusedLink(link);
460
461 [MethodImpl(MethodImplOptions.AggressiveInlining)]
462 protected virtual bool IsUnusedLink(TLink linkIndex)
463 {
464     if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
465         ↳ is not needed
466     {
467         ref var link = ref GetLinkReference(linkIndex);
468         return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
469     }
470 }

```



```

465     }
466     else
467     {
468         return true;
469     }
470 }
471
472 [MethodImpl(MethodImplOptions.AggressiveInlining)]
473 protected virtual TLink GetOne() => _one;
474
475 [MethodImpl(MethodImplOptions.AggressiveInlining)]
476 protected virtual TLink GetZero() => default;
477
478 [MethodImpl(MethodImplOptions.AggressiveInlining)]
479 protected virtual bool AreEqual(TLink first, TLink second) =>
480     ↪ _equalityComparer.Equals(first, second);
481
482 [MethodImpl(MethodImplOptions.AggressiveInlining)]
483 protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
484     ↪ second) < 0;
485
486 [MethodImpl(MethodImplOptions.AggressiveInlining)]
487 protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
488     ↪ _comparer.Compare(first, second) <= 0;
489
490 [MethodImpl(MethodImplOptions.AggressiveInlining)]
491 protected virtual bool GreaterThan(TLink first, TLink second) =>
492     ↪ _comparer.Compare(first, second) > 0;
493
494 [MethodImpl(MethodImplOptions.AggressiveInlining)]
495 protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
496     ↪ _comparer.Compare(first, second) >= 0;
497
498 [MethodImpl(MethodImplOptions.AggressiveInlining)]
499 protected virtual long ConvertToInt64(TLink value) =>
500     ↪ _addressToInt64Converter.Convert(value);
501
502 [MethodImpl(MethodImplOptions.AggressiveInlining)]
503 protected virtual TLink ConvertToAddress(long value) =>
504     ↪ _int64ToAddressConverter.Convert(value);
505
506 [MethodImpl(MethodImplOptions.AggressiveInlining)]
507 protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
508     ↪ second);
509
510 [MethodImpl(MethodImplOptions.AggressiveInlining)]
511 protected virtual TLink Subtract(TLink first, TLink second) =>
512     ↪ Arithmetic<TLink>.Subtract(first, second);
513
514 [MethodImpl(MethodImplOptions.AggressiveInlining)]
515 protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
516
517 [MethodImpl(MethodImplOptions.AggressiveInlining)]
518 protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
519
520 #region Disposable
521 protected override bool AllowMultipleDisposeCalls
522 {
523     [MethodImpl(MethodImplOptions.AggressiveInlining)]
524     get => true;
525 }
526
527 [MethodImpl(MethodImplOptions.AggressiveInlining)]
528 protected override void Dispose(bool manual, bool wasDisposed)
529 {
530     if (!wasDisposed)
531     {
532         ResetPointers();
533         _memory.DisposeIfPossible();
534     }
535 }
536
537 #endregion
538 }
539 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Methods.Lists;
3  using Platform.Converters;
4  using static System.Runtime.CompilerServices.Unsafe;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Memory.United.Generic
9  {
10     public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
11         ↳ ILinksListMethods<TLink>
12     {
13         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
14             ↳ UncheckedConverter<TLink, long>.Default;
15
16         private readonly byte* _links;
17         private readonly byte* _header;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnusedLinksListMethods(byte* links, byte* header)
21         {
22             _links = links;
23             _header = header;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
28             ↳ AsRef<LinksHeader<TLink>>(_header);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
32             ↳ AsRef<RawLink<TLink>>(_links + RawLink<TLink>.SizeInBytes *
33             ↳ _addressToInt64Converter.Convert(link));
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected override TLink GetPrevious(TLink element) => GetLinkReference(element).Source;
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override TLink GetNext(TLink element) => GetLinkReference(element).Target;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override TLink GetSize() => GetHeaderReference().FreeLinks;
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
52             ↳ element;
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
56             ↳ element;
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         protected override void SetPrevious(TLink element, TLink previous) =>
60             ↳ GetLinkReference(element).Source = previous;
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         protected override void SetNext(TLink element, TLink next) =>
64             ↳ GetLinkReference(element).Target = next;
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
68     }
69 }

```

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

```

```

9  {
10 public struct RawLink<TLink> : IEquatable<RawLink<TLink>>
11 {
12     private static readonly EqualityComparer<TLink> _equalityComparer =
13         ↳ EqualityComparer<TLink>.Default;
14
15     public static readonly long SizeInBytes = Structure<RawLink<TLink>>.Size;
16
17     public TLink Source;
18     public TLink Target;
19     public TLink LeftAsSource;
20     public TLink RightAsSource;
21     public TLink SizeAsSource;
22     public TLink LeftAsTarget;
23     public TLink RightAsTarget;
24     public TLink SizeAsTarget;
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     public override bool Equals(object obj) => obj is RawLink<TLink> link ? Equals(link) :
28         ↳ false;
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     public bool Equals(RawLink<TLink> other)
32     => _equalityComparer.Equals(Source, other.Source)
33     && _equalityComparer.Equals(Target, other.Target)
34     && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
35     && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
36     && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
37     && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
38     && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
39     && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     public override int GetHashCode() => (Source, Target, LeftAsSource, RightAsSource,
43         ↳ SizeAsSource, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     public static bool operator ==(RawLink<TLink> left, RawLink<TLink> right) =>
47         ↳ left.Equals(right);
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     public static bool operator !=(RawLink<TLink> left, RawLink<TLink> right) => !(left ==
51         ↳ right);
52 }
53 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.United.Generic;
3  using static System.Runtime.CompilerServices.Unsafe;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Memory.United.Specific
8  {
9      public unsafe abstract class UInt64LinksAvlBalancedTreeMethodsBase :
10         ↳ LinksAvlBalancedTreeMethodsBase<ulong>
11      {
12          protected new readonly RawLink<ulong>* Links;
13          protected new readonly LinksHeader<ulong>* Header;
14
15          protected UInt64LinksAvlBalancedTreeMethodsBase(LinksConstants<ulong> constants,
16         ↳ RawLink<ulong>* links, LinksHeader<ulong>* header)
17              : base(constants, (byte*)links, (byte*)header)
18          {
19              Links = links;
20              Header = header;
21          }
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override ulong GetZero() => 0UL;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override bool EqualToZero(ulong value) => value == 0UL;
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override bool AreEqual(ulong first, ulong second) => first == second;
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          protected override bool GreaterThanZero(ulong value) => value > 0UL;

```

```

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
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
44 ↪ always >= 0 for ulong
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
48
49 [MethodImpl(MethodImplOptions.AggressiveInlining)]
50 protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
51 ↪ for ulong
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected override bool LessThan(ulong first, ulong second) => first < second;
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 protected override ulong Increment(ulong value) => ++value;
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected override ulong Decrement(ulong value) => --value;
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override ulong Add(ulong first, ulong second) => first + second;
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override ulong Subtract(ulong first, ulong second) => first - second;
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
70 {
71     ref var firstLink = ref Links[first];
72     ref var secondLink = ref Links[second];
73     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
74 ↪ secondLink.Source, secondLink.Target);
75 }
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
79 {
80     ref var firstLink = ref Links[first];
81     ref var secondLink = ref Links[second];
82     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
83 ↪ secondLink.Source, secondLink.Target);
84 }
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 protected override ulong GetSizeValue(ulong value) => (value & 4294967264UL) >> 5;
88
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
91 ↪ storedValue & 31UL | (size & 134217727UL) << 5;
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 protected override bool GetLeftIsChildValue(ulong value) => (value & 16UL) >> 4 == 1UL;
95
96 [MethodImpl(MethodImplOptions.AggressiveInlining)]
97 protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
98 ↪ storedValue = storedValue & 4294967279UL | (As<bool, byte>(ref value) & 1UL) << 4;
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 protected override bool GetRightIsChildValue(ulong value) => (value & 8UL) >> 3 == 1UL;
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>
105 ↪ storedValue = storedValue & 4294967287UL | (As<bool, byte>(ref value) & 1UL) << 3;
106
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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 }

```

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

```

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 :
9     ↳ LinksSizeBalancedTreeMethodsBase<ulong>
10    {
11        protected new readonly RawLink<ulong>* Links;
12        protected new readonly LinksHeader<ulong>* Header;
13
14        protected UInt64LinksSizeBalancedTreeMethodsBase(LinksConstants<ulong> constants,
15        ↳ RawLink<ulong>* links, LinksHeader<ulong>* header)
16        : base(constants, (byte*)links, (byte*)header)
17        {
18            Links = links;
19            Header = header;
20        }
21
22        [MethodImpl(MethodImplOptions.AggressiveInlining)]
23        protected override ulong GetZero() => 0UL;
24
25        [MethodImpl(MethodImplOptions.AggressiveInlining)]
26        protected override bool EqualToZero(ulong value) => value == 0UL;
27
28        [MethodImpl(MethodImplOptions.AggressiveInlining)]
29        protected override bool AreEqual(ulong first, ulong second) => first == second;
30
31        [MethodImpl(MethodImplOptions.AggressiveInlining)]
32        protected override bool GreaterThanZero(ulong value) => value > 0UL;
33
34        [MethodImpl(MethodImplOptions.AggressiveInlining)]
35        protected override bool GreaterThan(ulong first, ulong second) => first > second;
36
37        [MethodImpl(MethodImplOptions.AggressiveInlining)]
38        protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
39
40        [MethodImpl(MethodImplOptions.AggressiveInlining)]
41        protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
42        ↳ always true for ulong
43
44        [MethodImpl(MethodImplOptions.AggressiveInlining)]
45        protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
46        ↳ always >= 0 for ulong
47
48        [MethodImpl(MethodImplOptions.AggressiveInlining)]
49        protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
50
51        [MethodImpl(MethodImplOptions.AggressiveInlining)]
52        protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
53        ↳ for ulong
54
55        [MethodImpl(MethodImplOptions.AggressiveInlining)]
56        protected override bool LessThan(ulong first, ulong second) => first < second;
57
58        [MethodImpl(MethodImplOptions.AggressiveInlining)]
59        protected override ulong Increment(ulong value) => ++value;
60
61        [MethodImpl(MethodImplOptions.AggressiveInlining)]
62        protected override ulong Decrement(ulong value) => --value;
63    }
64 }

```

```

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

```

1.52 ./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 :
8          ↪ UInt64LinksAvlBalancedTreeMethodsBase
9      {
10         public UInt64LinksSourcesAvlBalancedTreeMethods(LinksConstants<ulong> constants,
11             ↪ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
12             ↪ { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetLeftReference(ulong node) => ref
16             ↪ Links[node].LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20             ↪ Links[node].RightAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
30             ↪ left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
34             ↪ right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsSource);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
41             ↪ Links[node].SizeAsSource, size);
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override bool GetLeftIsChild(ulong node) =>
45             ↪ GetLeftIsChildValue(Links[node].SizeAsSource);
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override bool GetRightIsChild(ulong node) =>
49             ↪ GetRightIsChildValue(Links[node].SizeAsSource);
50     }
51 }

```

```

37
38 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 // protected override bool GetLeftIsChild(ulong node) => IsChild(node, GetLeft(node));
40
41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 protected override void SetLeftIsChild(ulong node, bool value) =>
43     ↳ SetLeftIsChildValue(ref Links[node].SizeAsSource, value);
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 protected override bool GetRightIsChild(ulong node) =>
47     ↳ GetRightIsChildValue(Links[node].SizeAsSource);
48
49 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
50 // protected override bool GetRightIsChild(ulong node) => IsChild(node, GetRight(node));
51
52 [MethodImpl(MethodImplOptions.AggressiveInlining)]
53 protected override void SetRightIsChild(ulong node, bool value) =>
54     ↳ SetRightIsChildValue(ref Links[node].SizeAsSource, value);
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 protected override sbyte GetBalance(ulong node) =>
58     ↳ GetBalanceValue(Links[node].SizeAsSource);
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
62     ↳ Links[node].SizeAsSource, value);
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 protected override ulong GetTreeRoot() => Header->RootAsSource;
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
72     ↳ ulong secondSource, ulong secondTarget)
73     => firstSource < secondSource || firstSource == secondSource && firstTarget <
74     ↳ secondTarget;
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
78     ↳ ulong secondSource, ulong secondTarget)
79     => firstSource > secondSource || firstSource == secondSource && firstTarget >
80     ↳ secondTarget;
81
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 protected override void ClearNode(ulong node)
84 {
85     ref var link = ref Links[node];
86     link.LeftAsSource = OUL;
87     link.RightAsSource = OUL;
88     link.SizeAsSource = OUL;
89 }
90 }
91 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Memory.United.Specific
6 {
7     public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
8         ↳ UInt64LinksSizeBalancedTreeMethodsBase
9     {
10         public UInt64LinksSourcesSizeBalancedTreeMethods(LinksConstants<ulong> constants,
11             ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
12             ↳ { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetLeftReference(ulong node) => ref
16             ↳ Links[node].LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20             ↳ Links[node].RightAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

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

```

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

```



```

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

```

1.55 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs

```

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

```

```

9      public UInt64LinksTargetsSizeBalancedTreeMethods(LinksConstants<ulong> constants,
10         ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
11         ↳ { }
12
13     [MethodImpl(MethodImplOptions.AggressiveInlining)]
14     protected override ref ulong GetLeftReference(ulong node) => ref
15         ↳ Links[node].LeftAsTarget;
16
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     protected override ref ulong GetRightReference(ulong node) => ref
19         ↳ Links[node].RightAsTarget;
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
29         ↳ left;
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
33         ↳ right;
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
40         ↳ size;
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override ulong GetTreeRoot() => Header->RootAsTarget;
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
50         ↳ ulong secondSource, ulong secondTarget)
51         ↳ => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
52         ↳ secondSource;
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
56         ↳ ulong secondSource, ulong secondTarget)
57         ↳ => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
58         ↳ secondSource;
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected override void ClearNode(ulong node)
62     {
63         ref var link = ref Links[node];
64         link.LeftAsTarget = OUL;
65         link.RightAsTarget = OUL;
66         link.SizeAsTarget = OUL;
67     }
68 }

```

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

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Memory;
4  using Platform.Singletons;
5  using Platform.Data.Doublets.Memory.United.Generic;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Memory.United.Specific
10 {
11     /// <summary>
12     /// <para>Represents a low-level implementation of direct access to resizable memory, for
13     ↳ organizing the storage of links with addresses represented as <see cref="ulong"
14     ↳ />.</para>

```

```

13  /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
    ↳ размером, для организации хранения связей с адресами представленными в виде <see
    ↳ cref="ulong"/>.</para>
14  /// </summary>
15  public unsafe class UInt64UnitedMemoryLinks : UnitedMemoryLinksBase<ulong>
16  {
17      private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
18      private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
19      private LinksHeader<ulong>* _header;
20      private RawLink<ulong>* _links;
21
22      [MethodImpl(MethodImplOptions.AggressiveInlining)]
23      public UInt64UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
24
25      /// <summary>
26      /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
27      ↳ минимальным шагом расширения базы данных.
28      /// </summary>
29      /// <param name="address">Полный путь к файлу базы данных.</param>
30      /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
31      ↳ байтах.</param>
32      [MethodImpl(MethodImplOptions.AggressiveInlining)]
33      public UInt64UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
34      ↳ FileMappedResizableDirectMemory(address, memoryReservationStep),
35      ↳ memoryReservationStep) { }
36
37      [MethodImpl(MethodImplOptions.AggressiveInlining)]
38      public UInt64UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
39      ↳ DefaultLinksSizeStep) { }
40
41      [MethodImpl(MethodImplOptions.AggressiveInlining)]
42      public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
43      ↳ memoryReservationStep) : this(memory, memoryReservationStep,
44      ↳ Default<LinksConstants<ulong>>.Instance, true) { }
45
46      [MethodImpl(MethodImplOptions.AggressiveInlining)]
47      public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
48      ↳ memoryReservationStep, LinksConstants<ulong> constants, bool useAvlBasedIndex) :
49      ↳ base(memory, memoryReservationStep, constants)
50      {
51          if (useAvlBasedIndex)
52          {
53              _createSourceTreeMethods = () => new
54              ↳ UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
55              _createTargetTreeMethods = () => new
56              ↳ UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
57          }
58          else
59          {
60              _createSourceTreeMethods = () => new
61              ↳ UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
62              _createTargetTreeMethods = () => new
63              ↳ UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
64          }
65          Init(memory, memoryReservationStep);
66      }
67
68      [MethodImpl(MethodImplOptions.AggressiveInlining)]
69      protected override void SetPointers(IResizableDirectMemory memory)
70      {
71          _header = (LinksHeader<ulong>*)memory.Pointer;
72          _links = (RawLink<ulong>*)memory.Pointer;
73          SourcesTreeMethods = _createSourceTreeMethods();
74          TargetsTreeMethods = _createTargetTreeMethods();
75          UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
76      }
77
78      [MethodImpl(MethodImplOptions.AggressiveInlining)]
79      protected override void ResetPointers()
80      {
81          base.ResetPointers();
82          _links = null;
83          _header = null;
84      }
85
86      [MethodImpl(MethodImplOptions.AggressiveInlining)]
87      protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
88
89
90
91
92
93
94
95
96
97
98
99

```

```

76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
    ↪     _links[linkIndex];
78
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     protected override bool AreEqual(ulong first, ulong second) => first == second;
81
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     protected override bool LessThan(ulong first, ulong second) => first < second;
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
87
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     protected override bool GreaterThan(ulong first, ulong second) => first > second;
90
91     [MethodImpl(MethodImplOptions.AggressiveInlining)]
92     protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
93
94     [MethodImpl(MethodImplOptions.AggressiveInlining)]
95     protected override ulong GetZero() => 0UL;
96
97     [MethodImpl(MethodImplOptions.AggressiveInlining)]
98     protected override ulong GetOne() => 1UL;
99
100    [MethodImpl(MethodImplOptions.AggressiveInlining)]
101    protected override long ConvertToInt64(ulong value) => (long)value;
102
103    [MethodImpl(MethodImplOptions.AggressiveInlining)]
104    protected override ulong ConvertToAddress(long value) => (ulong)value;
105
106    [MethodImpl(MethodImplOptions.AggressiveInlining)]
107    protected override ulong Add(ulong first, ulong second) => first + second;
108
109    [MethodImpl(MethodImplOptions.AggressiveInlining)]
110    protected override ulong Subtract(ulong first, ulong second) => first - second;
111
112    [MethodImpl(MethodImplOptions.AggressiveInlining)]
113    protected override ulong Increment(ulong link) => ++link;
114
115    [MethodImpl(MethodImplOptions.AggressiveInlining)]
116    protected override ulong Decrement(ulong link) => --link;
117 }
118 }

```

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

```

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 class UInt64UnusedLinksListMethods : UnusedLinksListMethods<ulong>
9      {
10         private readonly RawLink<ulong>* _links;
11         private readonly LinksHeader<ulong>* _header;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public UInt64UnusedLinksListMethods(RawLink<ulong>* links, LinksHeader<ulong>* header)
15             : base((byte*)links, (byte*)header)
16         {
17             _links = links;
18             _header = header;
19         }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref _links[link];
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
26     }
27 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Reflection;
3  using Platform.Converters;
4  using Platform.Numbers;
5  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 AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
12         ↳ IConverter<TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↳ EqualityComparer<TLink>.Default;
16         private static readonly TLink _zero = default;
17         private static readonly TLink _one = Arithmetic.Increment(_zero);
18
19         private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
23             ↳ powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
24             ↳ powerOf2ToUnaryNumberConverter;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Convert(TLink number)
28         {
29             var links = _links;
30             var nullConstant = links.Constants.Null;
31             var target = nullConstant;
32             for (var i = 0; !_equalityComparer.Equals(number, _zero) && i <
33                 ↳ NumericType<TLink>.BitsSize; i++)
34             {
35                 if (_equalityComparer.Equals(Bit.And(number, _one), _one))
36                 {
37                     target = _equalityComparer.Equals(target, nullConstant)
38                         ? _powerOf2ToUnaryNumberConverter.Convert(i)
39                         : links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
40                 }
41                 number = Bit.ShiftRight(number, 1);
42             }
43             return target;
44         }
45     }
46 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using Platform.Interfaces;
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 LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
12         ↳ IConverter<Doublet<TLink>, TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↳ EqualityComparer<TLink>.Default;
16
17         private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
18         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public LinkToItsFrequencyNumberConveter(
22             ILinks<TLink> links,
23             IProperty<TLink, TLink> frequencyPropertyOperator,
24             IConverter<TLink> unaryNumberToAddressConverter)
25             : base(links)
26         {
27             _frequencyPropertyOperator = frequencyPropertyOperator;
28             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public TLink Convert(Doublet<TLink> doublet)
33         {
34             var links = _links;
35             var link = links.SearchOrDefault(doublet.Source, doublet.Target);
36             if (_equalityComparer.Equals(link, default))
37             {

```

```

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 }

```

1.60 ./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>,
12         ↪ IConverter<int, TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↪ EqualityComparer<TLink>.Default;
16
17         private readonly TLink[] _unaryNumberPowersOf2;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
21         {
22             _unaryNumberPowersOf2 = new TLink[64];
23             _unaryNumberPowersOf2[0] = one;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Convert(int power)
28         {
29             Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
30                 ↪ - 1), nameof(power));
31             if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
32             {
33                 return _unaryNumberPowersOf2[power];
34             }
35             var previousPowerOf2 = Convert(power - 1);
36             var powerOf2 = _links.GetOrCreate(previousPowerOf2, previousPowerOf2);
37             _unaryNumberPowersOf2[power] = powerOf2;
38             return powerOf2;
39         }
40     }
41 }

```

1.61 ./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>,
11         ↪ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↪ EqualityComparer<TLink>.Default;
15         private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
16             ↪ UncheckedConverter<TLink, ulong>.Default;
17         private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
18             ↪ UncheckedConverter<ulong, TLink>.Default;
19         private static readonly TLink _zero = default;
20         private static readonly TLink _one = Arithmetic.Increment(_zero);
21
22         private readonly Dictionary<TLink, TLink> _unaryToUInt64;
23         private readonly TLink _unaryOne;
24     }
25 }

```

```

20
21 [MethodImpl(MethodImplOptions.AggressiveInlining)]
22 public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
23     : base(links)
24 {
25     _unaryOne = unaryOne;
26     _unaryToUInt64 = CreateUnaryToUInt64Dictionary(links, unaryOne);
27 }
28
29 [MethodImpl(MethodImplOptions.AggressiveInlining)]
30 public TLink Convert(TLink unaryNumber)
31 {
32     if (_equalityComparer.Equals(unaryNumber, default))
33     {
34         return default;
35     }
36     if (_equalityComparer.Equals(unaryNumber, _unaryOne))
37     {
38         return _one;
39     }
40     var links = _links;
41     var source = links.GetSource(unaryNumber);
42     var target = links.GetTarget(unaryNumber);
43     if (_equalityComparer.Equals(source, target))
44     {
45         return _unaryToUInt64[unaryNumber];
46     }
47     else
48     {
49         var result = _unaryToUInt64[source];
50         TLink lastValue;
51         while (!_unaryToUInt64.TryGetValue(target, out lastValue))
52         {
53             source = links.GetSource(target);
54             result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
55             target = links.GetTarget(target);
56         }
57         result = Arithmetic<TLink>.Add(result, lastValue);
58         return result;
59     }
60 }
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 private static Dictionary<TLink, TLink> CreateUnaryToUInt64Dictionary(ILinks<TLink>
↪ links, TLink unaryOne)
64 {
65     var unaryToUInt64 = new Dictionary<TLink, TLink>
66     {
67         { unaryOne, _one }
68     };
69     var unary = unaryOne;
70     var number = _one;
71     for (var i = 1; i < 64; i++)
72     {
73         unary = links.GetOrCreate(unary, unary);
74         number = Double(number);
75         unaryToUInt64.Add(unary, number);
76     }
77     return unaryToUInt64;
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 private static TLink Double(TLink number) =>
↪ _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
82 }
83 }

```

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

```

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

```

```

12 {
13     private static readonly EqualityComparer<TLink> _equalityComparer =
14         ↪ EqualityComparer<TLink>.Default;
15     private static readonly TLink _zero = default;
16     private static readonly TLink _one = Arithmetic.Increment(_zero);
17
18     private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
22         ↪ TLink> powerOf2ToUnaryNumberConverter) : base(links) => _unaryNumberPowerOf2Indicies
23         ↪ = CreateUnaryNumberPowerOf2IndiciesDictionary(powerOf2ToUnaryNumberConverter);
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     public TLink Convert(TLink sourceNumber)
27     {
28         var links = _links;
29         var nullConstant = links.Constants.Null;
30         var source = sourceNumber;
31         var target = nullConstant;
32         if (!_equalityComparer.Equals(source, nullConstant))
33         {
34             while (true)
35             {
36                 if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
37                 {
38                     SetBit(ref target, powerOf2Index);
39                     break;
40                 }
41                 else
42                 {
43                     powerOf2Index = _unaryNumberPowerOf2Indicies[links.GetSource(source)];
44                     SetBit(ref target, powerOf2Index);
45                     source = links.GetTarget(source);
46                 }
47             }
48         }
49         return target;
50     }
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     private static Dictionary<TLink, int>
54         ↪ CreateUnaryNumberPowerOf2IndiciesDictionary(IConverter<int, TLink>
55         ↪ powerOf2ToUnaryNumberConverter)
56     {
57         var unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
58         for (int i = 0; i < NumericType<TLink>.BitsSize; i++)
59         {
60             unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
61         }
62         return unaryNumberPowerOf2Indicies;
63     }
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     private static void SetBit(ref TLink target, int powerOf2Index) => target =
67         ↪ Bit.Or(target, Bit.ShiftLeft(_one, powerOf2Index));
68 }
69 }

```

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

```

1 using System.Linq;
2 using System.Collections.Generic;
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.PropertyOperators
9 {
10     public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>, IProperties<TLink, TLink,
11         ↪ TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↪ EqualityComparer<TLink>.Default;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public PropertiesOperator(ILinks<TLink> links) : base(links) { }
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         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 }

```

1.64 ./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 =
12             ↳ EqualityComparer<TLink>.Default;
13
14         private readonly TLink _propertyMarker;
15         private readonly TLink _propertyValueMarker;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
19             ↳ propertyValueMarker) : base(links)
20         {
21             _propertyMarker = propertyMarker;
22             _propertyValueMarker = propertyValueMarker;
23         }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public TLink Get(TLink link)
27         {
28             var property = _links.SearchOrDefault(link, _propertyMarker);
29             return GetValue(GetContainer(property));
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         private TLink GetContainer(TLink property)
34         {
35             var valueContainer = default(TLink);
36             if (_equalityComparer.Equals(property, default))
37             {
38                 return valueContainer;
39             }
40             var links = _links;
41             var constants = links.Constants;
42             var countinueConstant = constants.Continue;
43             var breakConstant = constants.Break;
44             var anyConstant = constants.Any;
45             var query = new Link<TLink>(anyConstant, property, anyConstant);
46             links.Each(candidate =>
47             {
48                 var candidateTarget = links.GetTarget(candidate);
49                 var valueTarget = links.GetTarget(candidateTarget);
50                 if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
51                 {
52                     valueContainer = links.GetIndex(candidate);
53                 }
54             });
55         }
56     }
57 }

```

```

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 }

```

1.65 ./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(ICollection<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
29                 ↪ generics) but will be possible with Sigil
30                 var halvedSequence = new TLink[(length / 2) + (length % 2)];
31                 HalveSequence(halvedSequence, sequence, length);
32                 sequence = halvedSequence;
33                 length = halvedSequence.Length;
34             }
35             // Keep creating layer after layer
36             while (length > 2)
37             {
38                 HalveSequence(sequence, sequence, length);
39                 length = (length / 2) + (length % 2);
40             }
41             return _links.GetOrCreate(sequence[0], sequence[1]);
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         private void HalveSequence(ICollection<TLink> destination, ICollection<TLink> source, int length)
45         {
46             var loopedLength = length - (length % 2);
47             for (var i = 0; i < loopedLength; i += 2)
48             {

```

```

49         destination[i / 2] = _links.GetOrCreate(source[i], source[i + 1]);
50     }
51     if (length > loopedLength)
52     {
53         destination[length / 2] = source[length - 1];
54     }
55 }
56 }
57 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections;
5  using Platform.Converters;
6  using Platform.Singletons;
7  using Platform.Numbers;
8  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.Sequences.Converters
13 {
14     /// <remarks>
15     /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от
16     ///     Links на этапе сжатия.
17     ///     А именно будет создаваться временный список пар необходимых для выполнения сжатия, в
18     ///     таком случае тип значения элемента массива может быть любым, как char так и ulong.
19     ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
20     ///     пар, а так же разом выполнить замену.
21     /// </remarks>
22     public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
23     {
24         private static readonly LinksConstants<TLink> _constants =
25             ↪ Default<LinksConstants<TLink>>.Instance;
26         private static readonly EqualityComparer<TLink> _equalityComparer =
27             ↪ EqualityComparer<TLink>.Default;
28         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
29
30         private static readonly TLink _zero = default;
31         private static readonly TLink _one = Arithmetic.Increment(_zero);
32
33         private readonly IConverter<IList<TLink>, TLink> _baseConverter;
34         private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
35         private readonly TLink _minFrequencyToCompress;
36         private readonly bool _doInitialFrequenciesIncrement;
37         private Doublet<TLink> _maxDoublet;
38         private LinkFrequency<TLink> _maxDoubletData;
39
40         private struct HalfDoublet
41         {
42             public TLink Element;
43             public LinkFrequency<TLink> DoubletData;
44
45             [MethodImpl(MethodImplOptions.AggressiveInlining)]
46             public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
47             {
48                 Element = element;
49                 DoubletData = doubletData;
50             }
51
52             public override string ToString() => $"{Element}: ({DoubletData})";
53         }
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
57             ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
58             : this(links, baseConverter, doubletFrequenciesCache, _one, true) { }
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
62             ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
63             ↪ doInitialFrequenciesIncrement)
64             : this(links, baseConverter, doubletFrequenciesCache, _one,
65                 ↪ doInitialFrequenciesIncrement) { }
66
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
69             ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
70             ↪ minFrequencyToCompress, bool doInitialFrequenciesIncrement)

```

```

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

```

```

136 [MethodImpl(MethodImplOptions.AggressiveInlining)]
137 private int ReplaceDoublets(HalfDoublet[] copy)
138 {
139     var oldLength = copy.Length;
140     var newLength = copy.Length;
141     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
142     {
143         var maxDoubletSource = _maxDoublet.Source;
144         var maxDoubletTarget = _maxDoublet.Target;
145         if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
146         {
147             _maxDoubletData.Link = _links.GetOrCreate(maxDoubletSource,
148                 ↪ maxDoubletTarget);
149         }
150         var maxDoubletReplacementLink = _maxDoubletData.Link;
151         oldLength--;
152         var oldLengthMinusTwo = oldLength - 1;
153         // Substitute all usages
154         int w = 0, r = 0; // (r == read, w == write)
155         for (; r < oldLength; r++)
156         {
157             if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
158                 ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
159             {
160                 if (r > 0)
161                 {
162                     var previous = copy[w - 1].Element;
163                     copy[w - 1].DoubletData.DecrementFrequency();
164                     copy[w - 1].DoubletData =
165                         ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
166                             ↪ maxDoubletReplacementLink);
167                 }
168                 if (r < oldLengthMinusTwo)
169                 {
170                     var next = copy[r + 2].Element;
171                     copy[r + 1].DoubletData.DecrementFrequency();
172                     copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
173                         ↪ next);
174                 }
175                 copy[w++] = copy[r];
176                 r++;
177                 newLength--;
178             }
179             else
180             {
181                 copy[w++] = copy[r];
182             }
183         }
184         if (w < newLength)
185         {
186             copy[w] = copy[r];
187         }
188         oldLength = newLength;
189         ResetMaxDoublet();
190         UpdateMaxDoublet(copy, newLength);
191     }
192     return newLength;
193 }
194
195 [MethodImpl(MethodImplOptions.AggressiveInlining)]
196 private void ResetMaxDoublet()
197 {
198     _maxDoublet = new Doublet<TLink>();
199     _maxDoubletData = new LinkFrequency<TLink>();
200 }
201
202 [MethodImpl(MethodImplOptions.AggressiveInlining)]
203 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
204 {
205     Doublet<TLink> doublet = default;
206     for (var i = 1; i < length; i++)
207     {
208         doublet.Source = copy[i - 1].Element;
209         doublet.Target = copy[i].Element;
210         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
211     }
212 }

```

```

209 [MethodImpl(MethodImplOptions.AggressiveInlining)]
210 private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
211 {
212     var frequency = data.Frequency;
213     var maxFrequency = _maxDoubletData.Frequency;
214     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
215     ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
216     ↪ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
217     ↪ _maxDoublet.Target)))
218     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
219     ↪ (_comparer.Compare(maxFrequency, frequency) < 0 ||
220     ↪ (_equalityComparer.Equals(maxFrequency, frequency) &&
221     ↪ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
222     ↪ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
223     ↪ better stability and better compression on sequent data and even on runderm
224     ↪ numbers data (but gives collisions anyway) */
225     {
226         _maxDoublet = doublet;
227         _maxDoubletData = data;
228     }
229 }
230 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Converters;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Sequences.Converters
8 {
9     public abstract class LinksListToSequenceConverterBase<TLink> : LinksOperatorBase<TLink>,
10     ↪ IConverter<IList<TLink>, TLink>
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         protected LinksListToSequenceConverterBase(ILinks<TLink> links) : base(links) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public abstract TLink Convert(IList<TLink> source);
17     }
18 }

```

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

```

1 using System.Collections.Generic;
2 using System.Linq;
3 using System.Runtime.CompilerServices;
4 using Platform.Converters;
5 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 =
15         ↪ EqualityComparer<TLink>.Default;
16         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
17
18         private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
22         ↪ sequenceToItsLocalElementLevelsConverter) : base(links)
23         => _sequenceToItsLocalElementLevelsConverter =
24         ↪ sequenceToItsLocalElementLevelsConverter;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public OptimalVariantConverter(ILinks<TLink> links, LinkFrequenciesCache<TLink>
28         ↪ linkFrequenciesCache)
29         : this(links, new SequenceToItsLocalElementLevelsConverter<TLink>(links, new Frequen
30         ↪ ciesCacheBasedLinkToItsFrequencyNumberConverter<TLink>(linkFrequenciesCache))) { }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public OptimalVariantConverter(ILinks<TLink> links)

```

```

29         : this(links, new LinkFrequenciesCache<TLink>(links, new
        ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links))) { }
30
31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 public override TLink Convert(ICollection<TLink> sequence)
33 {
34     var length = sequence.Count;
35     if (length == 1)
36     {
37         return sequence[0];
38     }
39     if (length == 2)
40     {
41         return _links.GetOrCreate(sequence[0], sequence[1]);
42     }
43     sequence = sequence.ToArray();
44     var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
45     while (length > 2)
46     {
47         var levelRepeat = 1;
48         var currentLevel = levels[0];
49         var previousLevel = levels[0];
50         var skipOnce = false;
51         var w = 0;
52         for (var i = 1; i < length; i++)
53         {
54             if (_equalityComparer.Equals(currentLevel, levels[i]))
55             {
56                 levelRepeat++;
57                 skipOnce = false;
58                 if (levelRepeat == 2)
59                 {
60                     sequence[w] = _links.GetOrCreate(sequence[i - 1], sequence[i]);
61                     var newLevel = i >= length - 1 ?
62                         GetPreviousLowerThanCurrentOrCurrent(previousLevel,
63                             ↪ currentLevel) :
64                         i < 2 ?
65                             GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
66                             GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
67                                 ↪ currentLevel, levels[i + 1]);
68                     levels[w] = newLevel;
69                     previousLevel = currentLevel;
70                     w++;
71                     levelRepeat = 0;
72                     skipOnce = true;
73                 }
74                 else if (i == length - 1)
75                 {
76                     sequence[w] = sequence[i];
77                     levels[w] = levels[i];
78                     w++;
79                 }
80             }
81             else
82             {
83                 currentLevel = levels[i];
84                 levelRepeat = 1;
85                 if (skipOnce)
86                 {
87                     skipOnce = false;
88                 }
89                 else
90                 {
91                     sequence[w] = sequence[i - 1];
92                     levels[w] = levels[i - 1];
93                     previousLevel = levels[w];
94                     w++;
95                 }
96                 if (i == length - 1)
97                 {
98                     sequence[w] = sequence[i];
99                     levels[w] = levels[i];
100                     w++;
101                 }
102             }
103         }
104         length = w;
105     }
106     return _links.GetOrCreate(sequence[0], sequence[1]);

```

```

105     }
106
107     [MethodImpl(MethodImplOptions.AggressiveInlining)]
108     private static TLink GetGreatestNeighbourLowerThanCurrentOrCurrent(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 }

```

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

```

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

```

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

```

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

```



```

14         public bool IsMatched(TLink argument) => _links.IsPartialPoint(argument);
15     }
16 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences.CriterionMatchers
8  {
9      public class MarkedSequenceCriterionMatcher<TLink> : ICriterionMatcher<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↳ EqualityComparer<TLink>.Default;
13
14         private readonly ILinks<TLink> _links;
15         private readonly TLink _sequenceMarkerLink;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
19         {
20             _links = links;
21             _sequenceMarkerLink = sequenceMarkerLink;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public bool IsMatched(TLink sequenceCandidate)
26         => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
27         || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
28             ↳ sequenceCandidate), _links.Constants.Null);
29     }
30 }

```

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

```

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>,
12         ↳ ISequenceAppender<TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↳ EqualityComparer<TLink>.Default;
16
17         private readonly IStack<TLink> _stack;
18         private readonly ISequenceHeightProvider<TLink> _heightProvider;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
22             ↳ ISequenceHeightProvider<TLink> heightProvider)
23             : base(links)
24         {
25             _stack = stack;
26             _heightProvider = heightProvider;
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public TLink Append(TLink sequence, TLink appendant)
31         {
32             var cursor = sequence;
33             var links = _links;
34             while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
35             {
36                 var source = links.GetSource(cursor);
37                 var target = links.GetTarget(cursor);
38                 if (_equalityComparer.Equals(_heightProvider.Get(source),
39                     ↳ _heightProvider.Get(target)))
40                 {
41                     break;
42                 }
43             }
44             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 }

```

1.73 ./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>>>>
13             ↪ _duplicateFragmentsProvider;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
17             ↪ IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
18             ↪ duplicateFragmentsProvider;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
22     }
23 }

```

1.74 ./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> :
19         ↪ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
20         ↪ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
21     {
22         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
23             ↪ UncheckedConverter<TLink, long>.Default;
24         private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
25             ↪ UncheckedConverter<TLink, ulong>.Default;
26         private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
27             ↪ UncheckedConverter<ulong, TLink>.Default;
28
29         private readonly ILinks<TLink> _links;
30         private readonly ILinks<TLink> _sequences;
31         private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
32         private BitString _visited;
33
34         private class ItemEqualityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
35             ↪ IList<TLink>>>
36         {
37             private readonly IListEqualityComparer<TLink> _listComparer;
38         }
39     }
40 }

```

```

33     public ItemEquilityComparer() => _listComparer =
34         ↳ Default<IListEqualityComparer<TLink>>.Instance;
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
38         ↳ KeyValuePair<IList<TLink>, IList<TLink>> right) =>
39         ↳ _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
40         ↳ right.Value);
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
44         ↳ (_listComparer.GetHashCode(pair.Key),
45         ↳ _listComparer.GetHashCode(pair.Value)).GetHashCode();
46 }
47
48 private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
49 {
50     private readonly IListComparer<TLink> _listComparer;
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
57         ↳ KeyValuePair<IList<TLink>, IList<TLink>> right)
58     {
59         var intermediateResult = _listComparer.Compare(left.Key, right.Key);
60         if (intermediateResult == 0)
61         {
62             intermediateResult = _listComparer.Compare(left.Value, right.Value);
63         }
64         return intermediateResult;
65     }
66 }
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
70     : base(minimumStringSegmentLength: 2)
71 {
72     _links = links;
73     _sequences = sequences;
74 }
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
78 {
79     _groups = new HashSet<KeyValuePair<IList<TLink>,
80         ↳ IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
81     var links = _links;
82     var count = links.Count();
83     _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
84     links.Each(link =>
85     {
86         var linkIndex = links.GetIndex(link);
87         var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
88         var constants = links.Constants;
89         if (!_visited.Get(linkBitIndex))
90         {
91             var sequenceElements = new List<TLink>();
92             var filler = new ListFiller<TLink, TLink>(sequenceElements, constants.Break);
93             _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
94                 ↳ LinkAddress<TLink>(linkIndex));
95             if (sequenceElements.Count > 2)
96             {
97                 WalkAll(sequenceElements);
98             }
99         }
100         return constants.Continue;
101     });
102     var resultList = _groups.ToList();
103     var comparer = Default<ItemComparer>.Instance;
104     resultList.Sort(comparer);
105
106     #if DEBUG
107     foreach (var item in resultList)
108     {
109         PrintDuplicates(item);
110     }
111
112     #endif
113     return resultList;

```

```

103     }
104
105     [MethodImpl(MethodImplOptions.AggressiveInlining)]
106     protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
        ↪ length) => new Segment<TLink>(elements, offset, length);
107
108     [MethodImpl(MethodImplOptions.AggressiveInlining)]
109     protected override void OnDuplicateFound(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 =
148                     ↪ _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
149                 duplicates.Add(sequenceIndex);
150             }
151         }
152         duplicates.Sort();
153         return duplicates;
154     }
155
156     [MethodImpl(MethodImplOptions.AggressiveInlining)]
157     private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
158     {
159         if (!(_links is ILinks<ulong> ulongLinks))
160         {
161             return;
162         }
163         var duplicatesKey = duplicatesItem.Key;
164         var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
165         Console.WriteLine($"{> {keyString} ({string.Join(", ", duplicatesKey)})");
166         var duplicatesList = duplicatesItem.Value;
167         for (int i = 0; i < duplicatesList.Count; i++)
168         {
169             var sequenceIndex = _addressToUInt64Converter.Convert(duplicatesList[i]);
170             var formattedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
                ↪ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
                ↪ UnicodeMap.IsCharLink(link.Index) ?
                ↪ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
171             Console.WriteLine(formattedSequenceStructure);
172             var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
                ↪ ulongLinks);

```

```

172         Console.WriteLine(sequenceString);
173     }
174     Console.WriteLine();
175 }
176 }
177 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5  using Platform.Numbers;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
10 {
11     /// <remarks>
12     /// Can be used to operate with many CompressingConverters (to keep global frequencies data
13     /// ↪ between them).
14     /// TODO: Extract interface to implement frequencies storage inside Links storage
15     /// </remarks>
16     public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
17     {
18         private static readonly EqualityComparer<TLink> _equalityComparer =
19             ↪ EqualityComparer<TLink>.Default;
20         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
21
22         private static readonly TLink _zero = default;
23         private static readonly TLink _one = Arithmetic.Increment(_zero);
24
25         private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
26         private readonly ICounter<TLink, TLink> _frequencyCounter;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
30             : base(links)
31         {
32             _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
33                 ↪ DoubletComparer<TLink>.Default);
34             _frequencyCounter = frequencyCounter;
35         }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
39         {
40             var doublet = new Doublet<TLink>(source, target);
41             return GetFrequency(ref doublet);
42         }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
46         {
47             _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
48             return data;
49         }
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         public void IncrementFrequencies(IList<TLink> sequence)
53         {
54             for (var i = 1; i < sequence.Count; i++)
55             {
56                 IncrementFrequency(sequence[i - 1], sequence[i]);
57             }
58         }
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
62         {
63             var doublet = new Doublet<TLink>(source, target);
64             return IncrementFrequency(ref doublet);
65         }
66
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         public void PrintFrequencies(IList<TLink> sequence)
69         {
70             for (var i = 1; i < sequence.Count; i++)
71             {
72                 PrintFrequency(sequence[i - 1], sequence[i]);
73             }
74         }
75     }
76 }

```

```

70     }
71 }
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 public void PrintFrequency(TLink source, TLink target)
75 {
76     var number = GetFrequency(source, target).Frequency;
77     Console.WriteLine("{0},{1}) - {2}", source, target, number);
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
82 {
83     if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
84     {
85         data.IncrementFrequency();
86     }
87     else
88     {
89         var link = _links.SearchOrDefault(doublet.Source, doublet.Target);
90         data = new LinkFrequency<TLink>(_one, link);
91         if (!_equalityComparer.Equals(link, default))
92         {
93             data.Frequency = Arithmetic.Add(data.Frequency,
94                 ↪ _frequencyCounter.Count(link));
95         }
96         _doubletsCache.Add(doublet, data);
97     }
98     return data;
99 }
100
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 public void ValidateFrequencies()
103 {
104     foreach (var entry in _doubletsCache)
105     {
106         var value = entry.Value;
107         var linkIndex = value.Link;
108         if (!_equalityComparer.Equals(linkIndex, default))
109         {
110             var frequency = value.Frequency;
111             var count = _frequencyCounter.Count(linkIndex);
112             // TODO: Why `frequency` always greater than `count` by 1?
113             if (((_comparer.Compare(frequency, count) > 0) &&
114                 ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count), _one) > 0))
115                 || ((_comparer.Compare(count, frequency) > 0) &&
116                 ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency), _one) > 0)))
117             {
118                 throw new InvalidOperationException("Frequencies validation failed.");
119             }
120             //else
121             //{
122             //     if (value.Frequency > 0)
123             //     {
124             //         var frequency = value.Frequency;
125             //         linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
126             //         var count = _countLinkFrequency(linkIndex);
127             //         if ((frequency > count && frequency - count > 1) || (count > frequency
128             //             ↪ && count - frequency > 1))
129             //             throw new InvalidOperationException("Frequencies validation
130             //             ↪ failed.");
131             //     }
132             // }
133 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Numbers;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
7 {

```

```

8     public class LinkFrequency<TLink>
9     {
10         public TLink Frequency { get; set; }
11         public TLink Link { get; set; }
12
13         [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 }

```

1.77 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToItsFrequencyValueConverter.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.Sequences.Frequencies.Cache
7 {
8     public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
9         ⇨ IConverter<Doublet<TLink>, TLink>
10     {
11         private readonly LinkFrequenciesCache<TLink> _cache;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public
15             ⇨ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
16             ⇨ cache) => _cache = cache;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
20     }
21 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Interfaces;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7 {
8     public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
9         ⇨ SequenceSymbolFrequencyOneOffCounter<TLink>
10     {
11         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
15             ⇨ ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
16             : base(links, sequenceLink, symbol)
17             => _markedSequenceMatcher = markedSequenceMatcher;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public override TLink Count()
21         {
22             if (!_markedSequenceMatcher.IsMatch(_sequenceLink))
23             {
24                 return default;
25             }
26             return base.Count();
27         }
28     }
29 }

```

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

```

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 =
14             ↳ EqualityComparer<TLink>.Default;
15         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
16
17         protected readonly ILinks<TLink> _links;
18         protected readonly TLink _sequenceLink;
19         protected readonly TLink _symbol;
20         protected TLink _total;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
24             ↳ TLink symbol)
25         {
26             _links = links;
27             _sequenceLink = sequenceLink;
28             _symbol = symbol;
29             _total = default;
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public virtual TLink Count()
34         {
35             if (_comparer.Compare(_total, default) > 0)
36             {
37                 return _total;
38             }
39             StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
40                 ↳ IsElement, VisitElement);
41             return _total;
42         }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
46             ↳ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
47             ↳ IsPartialPoint
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         private bool VisitElement(TLink element)
51         {
52             if (_equalityComparer.Equals(element, _symbol))
53             {
54                 _total = Arithmetic.Increment(_total);
55             }
56             return true;
57         }
58     }
59 }

```

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

```

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,
15             ↳ ICriterionMatcher<TLink> markedSequenceMatcher)
16         {
17             _links = links;
18             _markedSequenceMatcher = markedSequenceMatcher;
19         }
20     }
21 }

```



```

19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     public TLink Count(TLink argument) => new
21     ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
22     ↪ _markedSequenceMatcher, argument).Count();
23 }

```

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

```

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> :
10     ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>
11     {
12         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
16         ↪ ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
17         : base(links, symbol)
18         => _markedSequenceMatcher = markedSequenceMatcher;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override void CountSequenceSymbolFrequency(TLink link)
22         {
23             var symbolFrequencyCounter = new
24             ↪ MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
25             ↪ _markedSequenceMatcher, link, _symbol);
26             _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
27         }
28     }
29 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Interfaces;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.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
17         ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
18     }
19 }

```

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

```

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 =
13         ↪ EqualityComparer<TLink>.Default;
14         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
15
16         protected readonly ILinks<TLink> _links;
17         protected readonly TLink _symbol;
18         protected readonly HashSet<TLink> _visits;
19     }
20 }

```

```

18     protected TLink _total;
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
22     {
23         _links = links;
24         _symbol = symbol;
25         _visits = new HashSet<TLink>();
26         _total = default;
27     }
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public TLink Count()
31     {
32         if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
33         {
34             return _total;
35         }
36         CountCore(_symbol);
37         return _total;
38     }
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     private void CountCore(TLink link)
42     {
43         var any = _links.Constants.Any;
44         if (_equalityComparer.Equals(_links.Count(any, link), default))
45         {
46             CountSequenceSymbolFrequency(link);
47         }
48         else
49         {
50             _links.Each(EachElementHandler, any, link);
51         }
52     }
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected virtual void CountSequenceSymbolFrequency(TLink link)
56     {
57         var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
58             ↪ link, _symbol);
59         _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
60     }
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     private TLink EachElementHandler(IList<TLink> doublet)
64     {
65         var constants = _links.Constants;
66         var doubletIndex = doublet[constants.IndexPart];
67         if (_visits.Add(doubletIndex))
68         {
69             CountCore(doubletIndex);
70         }
71         return constants.Continue;
72     }
73 }

```

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

```

1  using System.Collections.Generic;
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.Sequences.HeightProviders
9  {
10     public class CachedSequenceHeightProvider<TLink> : ISequenceHeightProvider<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↪ EqualityComparer<TLink>.Default;
14
15         private readonly TLink _heightPropertyMarker;
16         private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
17         private readonly IConverter<TLink> _addressToUnaryNumberConverter;
18         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
19         private readonly IProperties<TLink, TLink, TLink> _propertyOperator;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

21     public CachedSequenceHeightProvider(
22         ISequenceHeightProvider<TLink> baseHeightProvider,
23         IConverter<TLink> addressToUnaryNumberConverter,
24         IConverter<TLink> unaryNumberToAddressConverter,
25         TLink heightPropertyMarker,
26         IProperties<TLink, TLink, TLink> propertyOperator)
27     {
28         _heightPropertyMarker = heightPropertyMarker;
29         _baseHeightProvider = baseHeightProvider;
30         _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
31         _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
32         _propertyOperator = propertyOperator;
33     }
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     public TLink Get(TLink sequence)
37     {
38         TLink height;
39         var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
40         if (_equalityComparer.Equals(heightValue, default))
41         {
42             height = _baseHeightProvider.Get(sequence);
43             heightValue = _addressToUnaryNumberConverter.Convert(height);
44             _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
45         }
46         else
47         {
48             height = _unaryNumberToAddressConverter.Convert(heightValue);
49         }
50         return height;
51     }
52 }
53 }

```

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

```

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.HeightProviders
8  {
9      public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
10         ↳ ISequenceHeightProvider<TLink>
11     {
12         private readonly ICriterionMatcher<TLink> _elementMatcher;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
16             ↳ elementMatcher) : base(links) => _elementMatcher = elementMatcher;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public TLink Get(TLink sequence)
20         {
21             var height = default(TLink);
22             var pairOrElement = sequence;
23             while (!_elementMatcher.IsMatched(pairOrElement))
24             {
25                 pairOrElement = _links.GetTarget(pairOrElement);
26                 height = Arithmetic.Increment(height);
27             }
28             return height;
29         }
30     }
31 }

```

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

```

1  using Platform.Interfaces;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.HeightProviders
6  {
7      public interface ISequenceHeightProvider<TLink> : IProvider<TLink, TLink>
8      {
9      }
10 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Sequences.Indexes
8 {
9     public class CachedFrequencyIncrementingSequenceIndex<TLink> : ISequenceIndex<TLink>
10    {
11        private static readonly EqualityComparer<TLink> _equalityComparer =
12            ↳ EqualityComparer<TLink>.Default;
13
14        private readonly LinkFrequenciesCache<TLink> _cache;
15
16        [MethodImpl(MethodImplOptions.AggressiveInlining)]
17        public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
18            ↳ _cache = cache;
19
20        [MethodImpl(MethodImplOptions.AggressiveInlining)]
21        public bool Add(IList<TLink> sequence)
22        {
23            var indexed = true;
24            var i = sequence.Count;
25            while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
26                ↳ { }
27            for (; i >= 1; i--)
28            {
29                _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
30            }
31            return indexed;
32        }
33
34        [MethodImpl(MethodImplOptions.AggressiveInlining)]
35        private bool IsIndexedWithIncrement(TLink source, TLink target)
36        {
37            var frequency = _cache.GetFrequency(source, target);
38            if (frequency == null)
39            {
40                return false;
41            }
42            var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
43            if (indexed)
44            {
45                _cache.IncrementFrequency(source, target);
46            }
47            return indexed;
48        }
49
50        [MethodImpl(MethodImplOptions.AggressiveInlining)]
51        public bool MightContain(IList<TLink> sequence)
52        {
53            var indexed = true;
54            var i = sequence.Count;
55            while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
56            return indexed;
57        }
58
59        [MethodImpl(MethodImplOptions.AggressiveInlining)]
60        private bool IsIndexed(TLink source, TLink target)
61        {
62            var frequency = _cache.GetFrequency(source, target);
63            if (frequency == null)
64            {
65                return false;
66            }
67            return !_equalityComparer.Equals(frequency.Frequency, default);
68        }
69    }
70 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Interfaces;
4 using Platform.Incrementers;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7

```

```

8 namespace Platform.Data.Doublets.Sequences.Indexes
9 {
10     public class FrequencyIncrementingSequenceIndex<TLink> : SequenceIndex<TLink>,
        ↳ ISequenceIndex<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
13
14         private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
15         private readonly IIncrementer<TLink> _frequencyIncrementer;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IProperty<TLink, TLink>
            ↳ frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
            : base(links)
19         {
20             _frequencyPropertyOperator = frequencyPropertyOperator;
21             _frequencyIncrementer = frequencyIncrementer;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public override bool Add(IList<TLink> sequence)
26         {
27             var indexed = true;
28             var i = sequence.Count;
29             while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
30                 ↳ { }
31             for (; i >= 1; i--)
32             {
33                 Increment(_links.GetOrCreate(sequence[i - 1], sequence[i]));
34             }
35             return indexed;
36         }
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         private bool IsIndexedWithIncrement(TLink source, TLink target)
40         {
41             var link = _links.SearchOrCreate(source, target);
42             var indexed = !_equalityComparer.Equals(link, default);
43             if (indexed)
44             {
45                 Increment(link);
46             }
47             return indexed;
48         }
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         private void Increment(TLink link)
52         {
53             var previousFrequency = _frequencyPropertyOperator.Get(link);
54             var frequency = _frequencyIncrementer.Increment(previousFrequency);
55             _frequencyPropertyOperator.Set(link, frequency);
56         }
57     }
58 }

```

1.89 ./csharp/Platform.Data.Doublets.Sequences.Indexes/ISequenceIndex.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 interface ISequenceIndex<TLink>
9     {
10         /// <summary>
11         /// Индексирует последовательность глобально, и возвращает значение,
12         /// определяющие была ли запрошенная последовательность проиндексирована ранее.
13         /// </summary>
14         /// <param name="sequence">Последовательность для индексации.</param>
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         bool Add(IList<TLink> sequence);
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         bool MightContain(IList<TLink> sequence);
20     }
21 }

```

1.90 ./csharp/Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.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 SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↪ EqualityComparer<TLink>.Default;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public SequenceIndex(ILinks<TLink> links) : base(links) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public virtual bool Add(IList<TLink> sequence)
18         {
19             var indexed = true;
20             var i = sequence.Count;
21             while (--i >= 1 && (indexed =
22                 ↪ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
23                 ↪ default))) { }
24             for (; i >= 1; i--)
25             {
26                 _links.GetOrCreate(sequence[i - 1], sequence[i]);
27             }
28             return indexed;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public virtual bool MightContain(IList<TLink> sequence)
33         {
34             var indexed = true;
35             var i = sequence.Count;
36             while (--i >= 1 && (indexed =
37                 ↪ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
38                 ↪ default))) { }
39             return indexed;
40         }
41     }
42 }
```

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

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

```

33         links.GetOrCreate(sequence[i - 1], sequence[i]);
34     }
35     });
36 }
37     return indexed;
38 }
39
40 [MethodImpl(MethodImplOptions.AggressiveInlining)]
41 public bool MightContain(ICollection<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 =
49             ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
50             ↳ sequence[i]), default))) { }
51         return indexed;
52     });
53 }

```

1.92 ./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(ICollection<TLink> sequence) => false;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public virtual bool MightContain(ICollection<TLink> sequence) => true;
15     }
16 }

```

1.93 ./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);
38             });
39         }
40     }
41 }

```

```

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
53                 ↳ меньше или равен stopAt");
54         }
55         #endif
56         if ((stopAt - startAt) == 0)
57         {
58             return new[] { sequence[startAt] };
59         }
60         if ((stopAt - startAt) == 1)
61         {
62             return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt]) };
63         }
64         var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
65         var last = 0;
66         for (var splitter = startAt; splitter < stopAt; splitter++)
67         {
68             var left = CreateAllVariants2Core(sequence, startAt, splitter);
69             var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
70             for (var i = 0; i < left.Length; i++)
71             {
72                 for (var j = 0; j < right.Length; j++)
73                 {
74                     var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
75                     if (variant == Constants.Null)
76                     {
77                         throw new NotImplementedException("Creation cancellation is not
78                             ↳ implemented.");
79                     }
80                     variants[last++] = variant;
81                 }
82             }
83         }
84         return variants;
85     }
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 public List<ulong> CreateAllVariants1(params ulong[] sequence)
89 {
90     return _sync.ExecuteWriteOperation(() =>
91     {
92         if (sequence.IsNullOrEmpty())
93         {
94             return new List<ulong>();
95         }
96         Links.Unsync.EnsureLinkExists(sequence);
97         if (sequence.Length == 1)
98         {
99             return new List<ulong> { sequence[0] };
100         }
101         var results = new
102             ↳ List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
103         return CreateAllVariants1Core(sequence, results);
104     });
105 }
106
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]
108 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
109 {
110     if (sequence.Length == 2)
111     {
112         var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
113         if (link == Constants.Null)
114         {

```



```

112         throw new NotImplementedException("Creation cancellation is not
113         ↪ implemented.");
114     }
115     results.Add(link);
116     return results;
117 }
118 var innerSequenceLength = sequence.Length - 1;
119 var innerSequence = new ulong[innerSequenceLength];
120 for (var li = 0; li < innerSequenceLength; li++)
121 {
122     var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
123     if (link == Constants.Null)
124     {
125         throw new NotImplementedException("Creation cancellation is not
126         ↪ implemented.");
127     }
128     for (var isi = 0; isi < li; isi++)
129     {
130         innerSequence[isi] = sequence[isi];
131     }
132     innerSequence[li] = link;
133     for (var isi = li + 1; isi < innerSequenceLength; isi++)
134     {
135         innerSequence[isi] = sequence[isi + 1];
136     }
137     CreateAllVariants1Core(innerSequence, results);
138 }
139 return results;
140 }
141 #endregion
142 [MethodImpl(MethodImplOptions.AggressiveInlining)]
143 public HashSet<ulong> Each1(params ulong[] sequence)
144 {
145     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
146     Each1(link =>
147     {
148         if (!visitedLinks.Contains(link))
149         {
150             visitedLinks.Add(link); // изучить почему случаются повторы
151         }
152         return true;
153     }, sequence);
154     return visitedLinks;
155 }
156 [MethodImpl(MethodImplOptions.AggressiveInlining)]
157 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
158 {
159     if (sequence.Length == 2)
160     {
161         Links.Unsync.Each(sequence[0], sequence[1], handler);
162     }
163     else
164     {
165         var innerSequenceLength = sequence.Length - 1;
166         for (var li = 0; li < innerSequenceLength; li++)
167         {
168             var left = sequence[li];
169             var right = sequence[li + 1];
170             if (left == 0 && right == 0)
171             {
172                 continue;
173             }
174             var linkIndex = li;
175             ulong[] innerSequence = null;
176             Links.Unsync.Each(doublet =>
177             {
178                 if (innerSequence == null)
179                 {
180                     innerSequence = new ulong[innerSequenceLength];
181                     for (var isi = 0; isi < linkIndex; isi++)
182                     {
183                         innerSequence[isi] = sequence[isi];
184                     }
185                     for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
186                     {

```

```

188         innerSequence[isi] = sequence[isi + 1];
189     }
190 }
191     innerSequence[linkIndex] = doublet[Constants.IndexPart];
192     Each1(handler, innerSequence);
193     return Constants.Continue;
194 }, Constants.Any, left, right);
195 }
196 }
197 }
198
199 [MethodImpl(MethodImplOptions.AggressiveInlining)]
200 public HashSet<ulong> EachPart(params ulong[] sequence)
201 {
202     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
203     EachPartCore(link =>
204     {
205         var linkIndex = link[Constants.IndexPart];
206         if (!visitedLinks.Contains(linkIndex))
207         {
208             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
209         }
210         return Constants.Continue;
211     }, sequence);
212     return visitedLinks;
213 }
214
215 [MethodImpl(MethodImplOptions.AggressiveInlining)]
216 public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
217 {
218     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
219     EachPartCore(link =>
220     {
221         var linkIndex = link[Constants.IndexPart];
222         if (!visitedLinks.Contains(linkIndex))
223         {
224             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
225             return handler(new LinkAddress<LinkIndex>(linkIndex));
226         }
227         return Constants.Continue;
228     }, sequence);
229 }
230
231 [MethodImpl(MethodImplOptions.AggressiveInlining)]
232 private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
233 ↪ sequence)
234 {
235     if (sequence.IsNullOrEmpty())
236     {
237         return;
238     }
239     Links.EnsureLinkIsAnyOrExists(sequence);
240     if (sequence.Length == 1)
241     {
242         var link = sequence[0];
243         if (link > 0)
244         {
245             handler(new LinkAddress<LinkIndex>(link));
246         }
247         else
248         {
249             Links.Each(Constants.Any, Constants.Any, handler);
250         }
251     }
252     else if (sequence.Length == 2)
253     {
254         //_links.Each(sequence[0], sequence[1], handler);
255         // o_|      x_o ...
256         // x_|      |__|
257         Links.Each(sequence[1], Constants.Any, doublet =>
258         {
259             var match = Links.SearchOrDefault(sequence[0], doublet);
260             if (match != Constants.Null)
261             {
262                 handler(new LinkAddress<LinkIndex>(match));
263             }
264             return true;
265         });
266     }

```

```

265 // |_x      ... x_o
266 // |_o      |__|
267 Links.Unsync.Each(Constants.Any, sequence[0], doublet =>
268 {
269     var match = Links.SearchOrDefault(doublet, sequence[1]);
270     if (match != 0)
271     {
272         handler(new LinkAddress<LinkIndex>(match));
273     }
274     return true;
275 });
276 //      .x o_.
277 //      |__|
278 PartialStepRight(x => handler(x), sequence[0], sequence[1]);
279 }
280 else
281 {
282     throw new NotImplementedException();
283 }
284 }
285
286 [MethodImpl(MethodImplOptions.AggressiveInlining)]
287 private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
288 {
289     Links.Unsync.Each(Constants.Any, left, doublet =>
290     {
291         StepRight(handler, doublet, right);
292         if (left != doublet)
293         {
294             PartialStepRight(handler, doublet, right);
295         }
296         return true;
297     });
298 }
299
300 [MethodImpl(MethodImplOptions.AggressiveInlining)]
301 private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
302 {
303     Links.Unsync.Each(left, Constants.Any, rightStep =>
304     {
305         TryStepRightUp(handler, right, rightStep);
306         return true;
307     });
308 }
309
310 [MethodImpl(MethodImplOptions.AggressiveInlining)]
311 private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
↪ stepFrom)
312 {
313     var upStep = stepFrom;
314     var firstSource = Links.Unsync.GetTarget(upStep);
315     while (firstSource != right && firstSource != upStep)
316     {
317         upStep = firstSource;
318         firstSource = Links.Unsync.GetSource(upStep);
319     }
320     if (firstSource == right)
321     {
322         handler(new LinkAddress<LinkIndex>(stepFrom));
323     }
324 }
325
326 // TODO: Test
327 [MethodImpl(MethodImplOptions.AggressiveInlining)]
328 private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
329 {
330     Links.Unsync.Each(right, Constants.Any, doublet =>
331     {
332         StepLeft(handler, left, doublet);
333         if (right != doublet)
334         {
335             PartialStepLeft(handler, left, doublet);
336         }
337         return true;
338     });
339 }
340
341 [MethodImpl(MethodImplOptions.AggressiveInlining)]
342 private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)

```

```

343 {
344     Links.Unsync.Each(Constants.Any, right, leftStep =>
345     {
346         TryStepLeftUp(handler, left, leftStep);
347         return true;
348     });
349 }
350
351 [MethodImpl(MethodImplOptions.AggressiveInlining)]
352 private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
353 {
354     var upStep = stepFrom;
355     var firstTarget = Links.Unsync.GetSource(upStep);
356     while (firstTarget != left && firstTarget != upStep)
357     {
358         upStep = firstTarget;
359         firstTarget = Links.Unsync.GetTarget(upStep);
360     }
361     if (firstTarget == left)
362     {
363         handler(new LinkAddress<LinkIndex>(stepFrom));
364     }
365 }
366
367 [MethodImpl(MethodImplOptions.AggressiveInlining)]
368 private bool StartsWith(ulong sequence, ulong link)
369 {
370     var upStep = sequence;
371     var firstSource = Links.Unsync.GetSource(upStep);
372     while (firstSource != link && firstSource != upStep)
373     {
374         upStep = firstSource;
375         firstSource = Links.Unsync.GetSource(upStep);
376     }
377     return firstSource == link;
378 }
379
380 [MethodImpl(MethodImplOptions.AggressiveInlining)]
381 private bool EndsWith(ulong sequence, ulong link)
382 {
383     var upStep = sequence;
384     var lastTarget = Links.Unsync.GetTarget(upStep);
385     while (lastTarget != link && lastTarget != upStep)
386     {
387         upStep = lastTarget;
388         lastTarget = Links.Unsync.GetTarget(upStep);
389     }
390     return lastTarget == link;
391 }
392
393 [MethodImpl(MethodImplOptions.AggressiveInlining)]
394 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
395 {
396     return _sync.ExecuteReadOperation(() =>
397     {
398         var results = new List<ulong>();
399         if (sequence.Length > 0)
400         {
401             Links.EnsureLinkExists(sequence);
402             var firstElement = sequence[0];
403             if (sequence.Length == 1)
404             {
405                 results.Add(firstElement);
406                 return results;
407             }
408             if (sequence.Length == 2)
409             {
410                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
411                 if (doublet != Constants.Null)
412                 {
413                     results.Add(doublet);
414                 }
415                 return results;
416             }
417             var linksInSequence = new HashSet<ulong>(sequence);
418             void handler(IList<LinkIndex> result)
419             {
420                 var resultIndex = result[Links.Constants.IndexPart];
421                 var filterPosition = 0;

```

```

422         StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
423         ↪ Links.Unsync.GetTarget,
424         ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
425         ↪ x =>
426         {
427             if (filterPosition == sequence.Length)
428             {
429                 filterPosition = -2; // Длиннее чем нужно
430                 return false;
431             }
432             if (x != sequence[filterPosition])
433             {
434                 filterPosition = -1;
435                 return false; // Начинается иначе
436             }
437             filterPosition++;
438             return true;
439         });
440     if (filterPosition == sequence.Length)
441     {
442         results.Add(resultIndex);
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],
456         ↪ sequence[sequence.Length - 1]);
457     }
458     }
459     return results;
460 }
461
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],
495                 ↪ sequence[i + 1]);

```

```

496         if (sequence.Length >= 3)
497         {
498             StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
499                 ↪ sequence[sequence.Length - 1]);
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)
509     ↪ => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
510
511 [MethodImpl(MethodImplOptions.AggressiveInlining)]
512 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
513     ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
514     ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
515     ↪ elementToString, insertComma, knownElements));
516
517 [MethodImpl(MethodImplOptions.AggressiveInlining)]
518 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
519     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
520     ↪ LinkIndex[] knownElements)
521 {
522     var linksInSequence = new HashSet<ulong>(knownElements);
523     //var entered = new HashSet<ulong>();
524     var sb = new StringBuilder();
525     sb.Append('{');
526     if (links.Exists(sequenceLink))
527     {
528         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
529             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
530             ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
531         {
532             if (insertComma && sb.Length > 1)
533             {
534                 sb.Append(',');
535             }
536             //if (entered.Contains(element))
537             //{
538             //    sb.Append('{');
539             //    elementToString(sb, element);
540             //    sb.Append('}');
541             //}
542             //else
543             elementToString(sb, element);
544             if (sb.Length < MaxSequenceFormatSize)
545             {
546                 return true;
547             }
548             sb.Append(insertComma ? ", ..." : "...");
549             return false;
550         });
551     }
552     sb.Append('}');
553     return sb.ToString();
554 }
555
556 [MethodImpl(MethodImplOptions.AggressiveInlining)]
557 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
558     ↪ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
559     ↪ knownElements);
560
561 [MethodImpl(MethodImplOptions.AggressiveInlining)]
562 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
563     ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
564     ↪ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
565     ↪ sequenceLink, elementToString, insertComma, knownElements));
566
567 [MethodImpl(MethodImplOptions.AggressiveInlining)]
568 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
569     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
570     ↪ LinkIndex[] knownElements)
571 {
572     var linksInSequence = new HashSet<ulong>(knownElements);

```

```

559     var entered = new HashSet<ulong>();
560     var sb = new StringBuilder();
561     sb.Append('{');
562     if (links.Exists(sequenceLink))
563     {
564         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
565             x => linksInSequence.Contains(x) || links.IsFullPoint(x),
566             entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
567             {
568                 if (insertComma && sb.Length > 1)
569                 {
570                     sb.Append(',');
571                 }
572                 if (entered.Contains(element))
573                 {
574                     sb.Append('{');
575                     elementToString(sb, element);
576                     sb.Append('}');
577                 }
578                 else
579                 {
580                     elementToString(sb, element);
581                 }
582                 if (sb.Length < MaxSequenceFormatSize)
583                 {
584                     return true;
585                 }
586                 sb.Append(insertComma ? ", ..." : "...");
587                 return false;
588             });
589     }
590     sb.Append('}');
591     return sb.ToString();
592 }
593 [MethodImpl(MethodImplOptions.AggressiveInlining)]
594 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
595 {
596     return _sync.ExecuteReadOperation(() =>
597     {
598         if (sequence.Length > 0)
599         {
600             Links.EnsureLinkExists(sequence);
601             var results = new HashSet<ulong>();
602             for (var i = 0; i < sequence.Length; i++)
603             {
604                 AllUsagesCore(sequence[i], results);
605             }
606             var filteredResults = new List<ulong>();
607             var linksInSequence = new HashSet<ulong>(sequence);
608             foreach (var result in results)
609             {
610                 var filterPosition = -1;
611                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
612                     Links.Unsync.GetTarget,
613                     x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
614                     x =>
615                     {
616                         if (filterPosition == (sequence.Length - 1))
617                         {
618                             return false;
619                         }
620                         if (filterPosition >= 0)
621                         {
622                             if (x == sequence[filterPosition + 1])
623                             {
624                                 filterPosition++;
625                             }
626                             else
627                             {
628                                 return false;
629                             }
630                         }
631                         if (filterPosition < 0)
632                         {
633                             if (x == sequence[0])
634                             {
635                                 filterPosition = 0;
636                             }
637                         }
638                     }
639             }
640             filteredResults.AddRange(results);
641         }
642     });
643 }

```

```

634         }
635     }
636     return true;
637 });
638 if (filterPosition == (sequence.Length - 1))
639 {
640     filteredResults.Add(result);
641 }
642 }
643 return filteredResults;
644 }
645 return new List<ulong>();
646 });
647 }
648
649 [MethodImpl(MethodImplOptions.AggressiveInlining)]
650 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
651 {
652     return _sync.ExecuteReadOperation(() =>
653     {
654         if (sequence.Length > 0)
655         {
656             Links.EnsureLinkExists(sequence);
657             var results = new HashSet<ulong>();
658             for (var i = 0; i < sequence.Length; i++)
659             {
660                 AllUsagesCore(sequence[i], results);
661             }
662             var filteredResults = new HashSet<ulong>();
663             var matcher = new Matcher(this, sequence, filteredResults, null);
664             matcher.AddAllPartialMatchedToResults(results);
665             return filteredResults;
666         }
667         return new HashSet<ulong>();
668     });
669 }
670
671 [MethodImpl(MethodImplOptions.AggressiveInlining)]
672 public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
673 ↪ params ulong[] sequence)
674 {
675     return _sync.ExecuteReadOperation(() =>
676     {
677         if (sequence.Length > 0)
678         {
679             Links.EnsureLinkExists(sequence);
680
681             var results = new HashSet<ulong>();
682             var filteredResults = new HashSet<ulong>();
683             var matcher = new Matcher(this, sequence, filteredResults, handler);
684             for (var i = 0; i < sequence.Length; i++)
685             {
686                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
687                 {
688                     return false;
689                 }
690             }
691             return true;
692         }
693         return true;
694     });
695 }
696
697 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
698 //{
699 //    return Sync.ExecuteReadOperation(() =>
700 //    {
701 //        if (sequence.Length > 0)
702 //        {
703 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
704 //
705 //            var firstResults = new HashSet<ulong>();
706 //            var lastResults = new HashSet<ulong>();
707 //
708 //            var first = sequence.First(x => x != LinksConstants.Any);
709 //            var last = sequence.Last(x => x != LinksConstants.Any);
710 //
711 //            AllUsagesCore(first, firstResults);
712 //            AllUsagesCore(last, lastResults);

```



```

712 //          firstResults.IntersectWith(lastResults);
713 //
714 //          //for (var i = 0; i < sequence.Length; i++)
715 //          //    AllUsagesCore(sequence[i], results);
716 //
717 //          var filteredResults = new HashSet<ulong>();
718 //          var matcher = new Matcher(this, sequence, filteredResults, null);
719 //          matcher.AddAllPartialMatchedToResults(firstResults);
720 //          return filteredResults;
721 //      }
722 //
723 //      return new HashSet<ulong>();
724 //  });
725 //}
726
727 [MethodImpl(MethodImplOptions.AggressiveInlining)]
728 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
729 {
730     return _sync.ExecuteReadOperation((Func<HashSet<ulong>>)(() =>
731     {
732         if (sequence.Length > 0)
733         {
734             ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links,
735                 ↪ (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,
756     ↪ IList<ulong> sequence)
757 {
758     return _sync.ExecuteReadOperation(() =>
759     {
760         if (sequence.Count > 0)
761         {
762             Links.EnsureLinkExists(sequence);
763             var results = new HashSet<LinkIndex>();
764             //var nextResults = new HashSet<ulong>();
765             //for (var i = 0; i < sequence.Length; i++)
766             //{
767             //    AllUsagesCore(sequence[i], nextResults);
768             //    if (results.IsNullOrEmpty())
769             //    {
770             //        results = nextResults;
771             //        nextResults = new HashSet<ulong>();
772             //    }
773             //    else
774             //    {
775             //        results.IntersectWith(nextResults);
776             //        nextResults.Clear();
777             //    }
778             //}
779             var collector1 = new AllUsagesCollector1(Links.Unsync, results);
780             collector1.Collect(Links.Unsync.GetLink(sequence[0]));
781             var next = new HashSet<ulong>();
782             for (var i = 1; i < sequence.Count; i++)
783             {
784                 var collector = new AllUsagesCollector1(Links.Unsync, next);
785                 collector.Collect(Links.Unsync.GetLink(sequence[i]));
786
787                 results.IntersectWith(next);
788                 next.Clear();

```

```

788     }
789     var filteredResults = new HashSet<ulong>();
790     var matcher = new Matcher(this, sequence, filteredResults, null,
791         ↪ readAsElements);
792     matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
793         ↪ x)); // OrderBy is a Hack
794     return filteredResults;
795 }
796 }
797
798 // Does not work
799 //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
800     ↪ params ulong[] sequence)
801 //{
802 //    var visited = new HashSet<ulong>();
803 //    var results = new HashSet<ulong>();
804 //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
805     ↪ true; }, readAsElements);
806 //    var last = sequence.Length - 1;
807 //    for (var i = 0; i < last; i++)
808 //    {
809 //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
810 //    }
811 //    return results;
812 //}
813
814 [MethodImpl(MethodImplOptions.AggressiveInlining)]
815 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
816 {
817     return _sync.ExecuteReadOperation(() =>
818     {
819         if (sequence.Length > 0)
820         {
821             Links.EnsureLinkExists(sequence);
822             //var firstElement = sequence[0];
823             //if (sequence.Length == 1)
824             //{
825             //    //results.Add(firstElement);
826             //    return results;
827             //}
828             //if (sequence.Length == 2)
829             //{
830             //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
831             //    //if (doublet != Doublets.Links.Null)
832             //    //    results.Add(doublet);
833             //    return results;
834             //}
835             //var lastElement = sequence[sequence.Length - 1];
836             //Func<ulong, bool> handler = x =>
837             //{
838             //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
839             //        ↪ results.Add(x);
840             //    return true;
841             //};
842             //if (sequence.Length >= 2)
843             //    StepRight(handler, sequence[0], sequence[1]);
844             //var last = sequence.Length - 2;
845             //for (var i = 1; i < last; i++)
846             //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
847             //if (sequence.Length >= 3)
848             //    StepLeft(handler, sequence[sequence.Length - 2],
849             //        ↪ sequence[sequence.Length - 1]);
850             //if (sequence.Length == 1)
851             //{
852             //    throw new NotImplementedException(); // all sequences, containing
853             //        ↪ this element?
854             //}
855             //if (sequence.Length == 2)
856             //{
857             //    var results = new List<ulong>();
858             //    PartialStepRight(results.Add, sequence[0], sequence[1]);
859             //    return results;
860             //}
861             //var matches = new List<List<ulong>>();
862             //var last = sequence.Length - 1;

```

```

858         for (var i = 0; i < last; i++)
859         {
860             var results = new List<ulong>();
861             //StepRight(results.Add, sequence[i], sequence[i + 1]);
862             PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
863             if (results.Count > 0)
864                 matches.Add(results);
865             else
866                 return results;
867             if (matches.Count == 2)
868             {
869                 var merged = new List<ulong>();
870                 for (var j = 0; j < matches[0].Count; j++)
871                     for (var k = 0; k < matches[1].Count; k++)
872                         CloseInnerConnections(merged.Add, matches[0][j],
873 → matches[1][k]);
874                 if (merged.Count > 0)
875                     matches = new List<List<ulong>> { merged };
876                 else
877                     return new List<ulong>();
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>() {
895             //    sequence[0] }; // or all sequences, containing this element?
896             //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
897             //    sequence[0], 1).ToList();
898             var results = new HashSet<ulong>();
899             foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
900             //    firstLinkUsages, 1))
901             {
902                 AllUsagesCore(match, results);
903             }
904             return results.ToList();
905         }
906     }
907     return new List<ulong>();
908 }
909
910 /// <remarks>
911 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
912 /// </remarks>
913 [MethodImpl(MethodImplOptions.AggressiveInlining)]
914 public HashSet<ulong> AllUsages(ulong link)
915 {
916     return _sync.ExecuteReadOperation(() =>
917     {
918         var usages = new HashSet<ulong>();
919         AllUsagesCore(link, usages);
920         return usages;
921     });
922 }
923
924 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
925 // той связи с которой начинался поиск (STTTSSSTT),
926 // причём достаточно одного бита для хранения перехода влево или вправо
927 [MethodImpl(MethodImplOptions.AggressiveInlining)]
928 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
929 {
930     bool handler(ulong doublet)
931     {
932         if (usages.Add(doublet))
933         {
934             AllUsagesCore(doublet, usages);
935         }
936     }
937     handler(link);
938 }

```

```

930         AllUsagesCore(doublet, usages);
931     }
932     return true;
933 }
934 Links.Unsync.Each(link, Constants.Any, handler);
935 Links.Unsync.Each(Constants.Any, link, handler);
936 }
937
938 [MethodImpl(MethodImplOptions.AggressiveInlining)]
939 public HashSet<ulong> AllBottomUsages(ulong link)
940 {
941     return _sync.ExecuteReadOperation(() =>
942     {
943         var visits = new HashSet<ulong>();
944         var usages = new HashSet<ulong>();
945         AllBottomUsagesCore(link, visits, usages);
946         return usages;
947     });
948 }
949
950 [MethodImpl(MethodImplOptions.AggressiveInlining)]
951 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
    ↳ usages)
952 {
953     bool handler(ulong doublet)
954     {
955         if (visits.Add(doublet))
956         {
957             AllBottomUsagesCore(doublet, visits, usages);
958         }
959         return true;
960     }
961     if (Links.Unsync.Count(Constants.Any, link) == 0)
962     {
963         usages.Add(link);
964     }
965     else
966     {
967         Links.Unsync.Each(link, Constants.Any, handler);
968         Links.Unsync.Each(Constants.Any, link, handler);
969     }
970 }
971
972 [MethodImpl(MethodImplOptions.AggressiveInlining)]
973 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
974 {
975     if (Options.UseSequenceMarker)
976     {
977         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
    ↳ Options.MarkedSequenceMatcher, symbol);
978         return counter.Count();
979     }
980     else
981     {
982         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
    ↳ symbol);
983         return counter.Count();
984     }
985 }
986
987 [MethodImpl(MethodImplOptions.AggressiveInlining)]
988 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
    ↳ LinkIndex> outerHandler)
989 {
990     bool handler(ulong doublet)
991     {
992         if (usages.Add(doublet))
993         {
994             if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
995             {
996                 return false;
997             }
998             if (!AllUsagesCore1(doublet, usages, outerHandler))
999             {
1000                 return false;
1001             }
1002         }
1003         return true;

```

```

1004     }
1005     return Links.Unsync.Each(link, Constants.Any, handler)
1006         && Links.Unsync.Each(Constants.Any, link, handler);
1007 }
1008
1009 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1010 public void CalculateAllUsages(ulong[] totals)
1011 {
1012     var calculator = new AllUsagesCalculator(Links, totals);
1013     calculator.Calculate();
1014 }
1015
1016 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1017 public void CalculateAllUsages2(ulong[] totals)
1018 {
1019     var calculator = new AllUsagesCalculator2(Links, totals);
1020     calculator.Calculate();
1021 }
1022
1023 private class AllUsagesCalculator
1024 {
1025     private readonly SynchronizedLinks<ulong> _links;
1026     private readonly ulong[] _totals;
1027
1028     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1029     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
1030     {
1031         _links = links;
1032         _totals = totals;
1033     }
1034
1035     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1036     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1037         ↪ CalculateCore);
1038
1039     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1040     private bool CalculateCore(ulong link)
1041     {
1042         if (_totals[link] == 0)
1043         {
1044             var total = 1UL;
1045             _totals[link] = total;
1046             var visitedChildren = new HashSet<ulong>();
1047             bool linkCalculator(ulong child)
1048             {
1049                 if (link != child && visitedChildren.Add(child))
1050                 {
1051                     total += _totals[child] == 0 ? 1 : _totals[child];
1052                 }
1053                 return true;
1054             }
1055             _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1056             _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1057             _totals[link] = total;
1058         }
1059         return true;
1060     }
1061 }
1062
1063 private class AllUsagesCalculator2
1064 {
1065     private readonly SynchronizedLinks<ulong> _links;
1066     private readonly ulong[] _totals;
1067
1068     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1069     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1070     {
1071         _links = links;
1072         _totals = totals;
1073     }
1074
1075     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1076     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1077         ↪ CalculateCore);
1078
1079     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1080     private bool IsElement(ulong link)
1081     {
1082         // _linksInSequence.Contains(link) ||

```

```

1081         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1082             ↪ link;
1083     }
1084     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1085     private bool CalculateCore(ulong link)
1086     {
1087         // TODO: Проработать защиту от заикливания
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             _totals[link]++;
1144             return true;
1145         }
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;

```

```

1159     }
1160
1161     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1162     public bool Collect(ulong link)
1163     {
1164         if (_usages.Add(link))
1165         {
1166             _links.Each(link, _links.Constants.Any, Collect);
1167             _links.Each(_links.Constants.Any, link, Collect);
1168         }
1169         return true;
1170     }
1171 }
1172
1173 private class AllUsagesCollector1
1174 {
1175     private readonly ILinks<ulong> _links;
1176     private readonly HashSet<ulong> _usages;
1177     private readonly ulong _continue;
1178
1179     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1180     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1181     {
1182         _links = links;
1183         _usages = usages;
1184         _continue = _links.Constants.Continue;
1185     }
1186
1187     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1188     public ulong Collect(IList<ulong> link)
1189     {
1190         var linkIndex = _links.GetIndex(link);
1191         if (_usages.Add(linkIndex))
1192         {
1193             _links.Each(Collect, _links.Constants.Any, linkIndex);
1194         }
1195         return _continue;
1196     }
1197 }
1198
1199 private class AllUsagesCollector2
1200 {
1201     private readonly ILinks<ulong> _links;
1202     private readonly BitString _usages;
1203
1204     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1205     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1206     {
1207         _links = links;
1208         _usages = usages;
1209     }
1210
1211     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1212     public bool Collect(ulong link)
1213     {
1214         if (_usages.Add((long)link))
1215         {
1216             _links.Each(link, _links.Constants.Any, Collect);
1217             _links.Each(_links.Constants.Any, link, Collect);
1218         }
1219         return true;
1220     }
1221 }
1222
1223 private class AllUsagesIntersectingCollector
1224 {
1225     private readonly SynchronizedLinks<ulong> _links;
1226     private readonly HashSet<ulong> _intersectWith;
1227     private readonly HashSet<ulong> _usages;
1228     private readonly HashSet<ulong> _enter;
1229
1230     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1231     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
↪ intersectWith, HashSet<ulong> usages)
1232     {
1233         _links = links;
1234         _intersectWith = intersectWith;
1235         _usages = usages;
1236         _enter = new HashSet<ulong>(); // защита от заикливания
1237     }

```

```

1238 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1239 public bool Collect(ulong link)
1240 {
1241     if (_enter.Add(link))
1242     {
1243         if (_intersectWith.Contains(link))
1244         {
1245             _usages.Add(link);
1246         }
1247         _links.Unsync.Each(link, _links.Constants.Any, Collect);
1248         _links.Unsync.Each(_links.Constants.Any, link, Collect);
1249     }
1250     return true;
1251 }
1252 }
1253
1254 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1255 private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
1256 ↪ right)
1257 {
1258     TryStepLeftUp(handler, left, right);
1259     TryStepRightUp(handler, right, left);
1260 }
1261
1262 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1263 private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
1264 ↪ right)
1265 {
1266     // Direct
1267     if (left == right)
1268     {
1269         handler(new LinkAddress<LinkIndex>(left));
1270     }
1271     var doublet = Links.Unsync.SearchOrDefault(left, right);
1272     if (doublet != Constants.Null)
1273     {
1274         handler(new LinkAddress<LinkIndex>(doublet));
1275     }
1276     // Inner
1277     CloseInnerConnections(handler, left, right);
1278     // Outer
1279     StepLeft(handler, left, right);
1280     StepRight(handler, left, right);
1281     PartialStepRight(handler, left, right);
1282     PartialStepLeft(handler, left, right);
1283 }
1284
1285 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1286 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
1287 ↪ HashSet<ulong> previousMatchings, long startAt)
1288 {
1289     if (startAt >= sequence.Length) // ?
1290     {
1291         return previousMatchings;
1292     }
1293     var secondLinkUsages = new HashSet<ulong>();
1294     AllUsagesCore(sequence[startAt], secondLinkUsages);
1295     secondLinkUsages.Add(sequence[startAt]);
1296     var matchings = new HashSet<ulong>();
1297     var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
1298     //for (var i = 0; i < previousMatchings.Count; i++)
1299     foreach (var secondLinkUsage in secondLinkUsages)
1300     {
1301         foreach (var previousMatching in previousMatchings)
1302         {
1303             //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
1304             ↪ secondLinkUsage);
1305             StepRight(filler.AddFirstAndReturnConstant, previousMatching,
1306             ↪ secondLinkUsage);
1307             TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
1308             ↪ previousMatching);
1309             //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1310             ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1311             ↪ желаемым результатам.
1312             PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
1313             ↪ secondLinkUsage);

```



```

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>();

```

```

1379         AllUsagesCore(linksToConnect[i], next);
1380         results.IntersectWith(next);
1381     }
1382 }
1383     return results;
1384 });
1385 }
1386
1387 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1388 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1389 {
1390     return _sync.ExecuteReadOperation(() =>
1391     {
1392         var results = new HashSet<ulong>();
1393         if (linksToConnect.Length > 0)
1394         {
1395             Links.EnsureLinkExists(linksToConnect);
1396             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1397             collector1.Collect(linksToConnect[0]);
1398             var next = new HashSet<ulong>();
1399             for (var i = 1; i < linksToConnect.Length; i++)
1400             {
1401                 var collector = new AllUsagesCollector(Links.Unsync, next);
1402                 collector.Collect(linksToConnect[i]);
1403                 results.IntersectWith(next);
1404                 next.Clear();
1405             }
1406             return results;
1407         }
1408     });
1409 }
1410
1411 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1412 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1413 {
1414     return _sync.ExecuteReadOperation(() =>
1415     {
1416         var results = new HashSet<ulong>();
1417         if (linksToConnect.Length > 0)
1418         {
1419             Links.EnsureLinkExists(linksToConnect);
1420             var collector1 = new AllUsagesCollector(Links, results);
1421             collector1.Collect(linksToConnect[0]);
1422             //AllUsagesCore(linksToConnect[0], results);
1423             for (var i = 1; i < linksToConnect.Length; i++)
1424             {
1425                 var next = new HashSet<ulong>();
1426                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1427                 collector.Collect(linksToConnect[i]);
1428                 //AllUsagesCore(linksToConnect[i], next);
1429                 //results.IntersectWith(next);
1430                 results = next;
1431             }
1432             return results;
1433         }
1434     });
1435 }
1436
1437 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1438 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1439 {
1440     return _sync.ExecuteReadOperation(() =>
1441     {
1442         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1443         ↪ BitArray((int)_links.Total + 1);
1444         if (linksToConnect.Length > 0)
1445         {
1446             Links.EnsureLinkExists(linksToConnect);
1447             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1448             collector1.Collect(linksToConnect[0]);
1449             for (var i = 1; i < linksToConnect.Length; i++)
1450             {
1451                 var next = new BitString((long)Links.Unsync.Count() + 1); //new
1452                 ↪ BitArray((int)_links.Total + 1);
1453                 var collector = new AllUsagesCollector2(Links.Unsync, next);
1454                 collector.Collect(linksToConnect[i]);
1455                 results = results.And(next);
1456             }
1457         }
1458     });
1459 }

```

```

1455     }
1456     return results.GetSetUInt64Indices();
1457 });
1458 }
1459
1460 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1461 private static ulong[] Simplify(ulong[] sequence)
1462 {
1463     // Считаем новый размер последовательности
1464     long newLength = 0;
1465     var zeroOrManyStepped = false;
1466     for (var i = 0; i < sequence.Length; i++)
1467     {
1468         if (sequence[i] == ZeroOrMany)
1469         {
1470             if (zeroOrManyStepped)
1471             {
1472                 continue;
1473             }
1474             zeroOrManyStepped = true;
1475         }
1476         else
1477         {
1478             //if (zeroOrManyStepped) Is it efficient?
1479             zeroOrManyStepped = false;
1480         }
1481         newLength++;
1482     }
1483     // Строим новую последовательность
1484     zeroOrManyStepped = false;
1485     var newSequence = new ulong[newLength];
1486     long j = 0;
1487     for (var i = 0; i < sequence.Length; i++)
1488     {
1489         //var current = zeroOrManyStepped;
1490         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1491         //if (current && zeroOrManyStepped)
1492         //    continue;
1493         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1494         //if (zeroOrManyStepped && newZeroOrManyStepped)
1495         //    continue;
1496         //zeroOrManyStepped = newZeroOrManyStepped;
1497         if (sequence[i] == ZeroOrMany)
1498         {
1499             if (zeroOrManyStepped)
1500             {
1501                 continue;
1502             }
1503             zeroOrManyStepped = true;
1504         }
1505         else
1506         {
1507             //if (zeroOrManyStepped) Is it efficient?
1508             zeroOrManyStepped = false;
1509         }
1510         newSequence[j++] = sequence[i];
1511     }
1512     return newSequence;
1513 }
1514
1515 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1516 public static void TestSimplify()
1517 {
1518     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
1519     ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1520     var simplifiedSequence = Simplify(sequence);
1521 }
1522
1523 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1524 public List<ulong> GetSimilarSequences() => new List<ulong>();
1525
1526 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1527 public void Prediction()
1528 {
1529     //_links
1530     //sequences
1531 }
1532
1533 #region From Triplets

```

```

1533 //public static void DeleteSequence(Link sequence)
1534 //{
1535 //}
1536
1537
1538 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1539 public List<ulong> CollectMatchingSequences(ulong[] links)
1540 {
1541     if (links.Length == 1)
1542     {
1543         throw new InvalidOperationException("Подпоследовательности с одним элементом не
1544             ↳ поддерживаются.");
1545     }
1546     var leftBound = 0;
1547     var rightBound = links.Length - 1;
1548     var left = links[leftBound++];
1549     var right = links[rightBound--];
1550     var results = new List<ulong>();
1551     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1552     return results;
1553 }
1554
1555 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1556 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1557     ↳ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1558 {
1559     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1560     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1561     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1562     {
1563         var nextLeftLink = middleLinks[leftBound];
1564         var elements = GetRightElements(leftLink, nextLeftLink);
1565         if (leftBound <= rightBound)
1566         {
1567             for (var i = elements.Length - 1; i >= 0; i--)
1568             {
1569                 var element = elements[i];
1570                 if (element != 0)
1571                 {
1572                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
1573                         ↳ rightLink, rightBound, ref results);
1574                 }
1575             }
1576         }
1577         else
1578         {
1579             for (var i = elements.Length - 1; i >= 0; i--)
1580             {
1581                 var element = elements[i];
1582                 if (element != 0)
1583                 {
1584                     results.Add(element);
1585                 }
1586             }
1587         }
1588     }
1589     else
1590     {
1591         var nextRightLink = middleLinks[rightBound];
1592         var elements = GetLeftElements(rightLink, nextRightLink);
1593         if (leftBound <= rightBound)
1594         {
1595             for (var i = elements.Length - 1; i >= 0; i--)
1596             {
1597                 var element = elements[i];
1598                 if (element != 0)
1599                 {
1600                     CollectMatchingSequences(leftLink, leftBound, middleLinks,
1601                         ↳ elements[i], rightBound - 1, ref results);
1602                 }
1603             }
1604         }
1605         else
1606         {
1607             for (var i = elements.Length - 1; i >= 0; i--)
1608             {
1609                 var element = elements[i];
1610                 if (element != 0)

```

```

1607         {
1608             results.Add(element);
1609         }
1610     }
1611 }
1612
1613 }
1614
1615 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1616 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1617 {
1618     var result = new ulong[5];
1619     TryStepRight(startLink, rightLink, result, 0);
1620     Links.Each(Constants.Any, startLink, couple =>
1621     {
1622         if (couple != startLink)
1623         {
1624             if (TryStepRight(couple, rightLink, result, 2))
1625             {
1626                 return false;
1627             }
1628         }
1629         return true;
1630     });
1631     if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1632     {
1633         result[4] = startLink;
1634     }
1635     return result;
1636 }
1637
1638 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1639 public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1640 {
1641     var added = 0;
1642     Links.Each(startLink, Constants.Any, couple =>
1643     {
1644         if (couple != startLink)
1645         {
1646             var coupleTarget = Links.GetTarget(couple);
1647             if (coupleTarget == rightLink)
1648             {
1649                 result[offset] = couple;
1650                 if (++added == 2)
1651                 {
1652                     return false;
1653                 }
1654             }
1655             else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
1656                 ↪ == Net.And &&
1657             {
1658                 result[offset + 1] = couple;
1659                 if (++added == 2)
1660                 {
1661                     return false;
1662                 }
1663             }
1664         }
1665         return true;
1666     });
1667     return added > 0;
1668 }
1669
1670 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1671 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1672 {
1673     var result = new ulong[5];
1674     TryStepLeft(startLink, leftLink, result, 0);
1675     Links.Each(startLink, Constants.Any, couple =>
1676     {
1677         if (couple != startLink)
1678         {
1679             if (TryStepLeft(couple, leftLink, result, 2))
1680             {
1681                 return false;
1682             }
1683         }
1684         return true;
1685     });

```

```

1685         if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1686         {
1687             result[4] = leftLink;
1688         }
1689         return result;
1690     }
1691
1692     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1693     public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1694     {
1695         var added = 0;
1696         Links.Each(Constants.Any, startLink, couple =>
1697         {
1698             if (couple != startLink)
1699             {
1700                 var coupleSource = Links.GetSource(couple);
1701                 if (coupleSource == leftLink)
1702                 {
1703                     result[offset] = couple;
1704                     if (++added == 2)
1705                     {
1706                         return false;
1707                     }
1708                 }
1709                 else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1710                     ↪ == Net.And &&
1711                 {
1712                     result[offset + 1] = couple;
1713                     if (++added == 2)
1714                     {
1715                         return false;
1716                     }
1717                 }
1718             }
1719             return true;
1720         });
1721         return added > 0;
1722     }
1723
1724 #endregion
1725
1726 #region Walkers
1727
1728 public class PatternMatcher : RightSequenceWalker<ulong>
1729 {
1730     private readonly Sequences _sequences;
1731     private readonly ulong[] _patternSequence;
1732     private readonly HashSet<LinkIndex> _linksInSequence;
1733     private readonly HashSet<LinkIndex> _results;
1734
1735     #region Pattern Match
1736
1737     enum PatternBlockType
1738     {
1739         Undefined,
1740         Gap,
1741         Elements
1742     }
1743
1744     struct PatternBlock
1745     {
1746         public PatternBlockType Type;
1747         public long Start;
1748         public long Stop;
1749     }
1750
1751     private readonly List<PatternBlock> _pattern;
1752     private int _patternPosition;
1753     private long _sequencePosition;
1754
1755 #endregion
1756
1757     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1758     public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1759         ↪ HashSet<LinkIndex> results)
1760         : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1761     {
1762         _sequences = sequences;
1763         _patternSequence = patternSequence;
1764         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1765             ↪ _sequences.Constants.Any && x != ZeroOrMany));

```

```

1763         _results = results;
1764         _pattern = CreateDetailedPattern();
1765     }
1766
1767     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1768     protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
        ↳ base.IsElement(link);
1769
1770     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1771     public bool PatternMatch(LinkIndex sequenceToMatch)
1772     {
1773         _patternPosition = 0;
1774         _sequencePosition = 0;
1775         foreach (var part in Walk(sequenceToMatch))
1776         {
1777             if (!PatternMatchCore(part))
1778             {
1779                 break;
1780             }
1781         }
1782         return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
        ↳ - 1 && _pattern[_patternPosition].Start == 0);
1783     }
1784
1785     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1786     private List<PatternBlock> CreateDetailedPattern()
1787     {
1788         var pattern = new List<PatternBlock>();
1789         var patternBlock = new PatternBlock();
1790         for (var i = 0; i < _patternSequence.Length; i++)
1791         {
1792             if (patternBlock.Type == PatternBlockType.Undefined)
1793             {
1794                 if (_patternSequence[i] == _sequences.Constants.Any)
1795                 {
1796                     patternBlock.Type = PatternBlockType.Gap;
1797                     patternBlock.Start = 1;
1798                     patternBlock.Stop = 1;
1799                 }
1800                 else if (_patternSequence[i] == ZeroOrMany)
1801                 {
1802                     patternBlock.Type = PatternBlockType.Gap;
1803                     patternBlock.Start = 0;
1804                     patternBlock.Stop = long.MaxValue;
1805                 }
1806                 else
1807                 {
1808                     patternBlock.Type = PatternBlockType.Elements;
1809                     patternBlock.Start = i;
1810                     patternBlock.Stop = i;
1811                 }
1812             }
1813             else if (patternBlock.Type == PatternBlockType.Elements)
1814             {
1815                 if (_patternSequence[i] == _sequences.Constants.Any)
1816                 {
1817                     pattern.Add(patternBlock);
1818                     patternBlock = new PatternBlock
1819                     {
1820                         Type = PatternBlockType.Gap,
1821                         Start = 1,
1822                         Stop = 1
1823                     };
1824                 }
1825                 else if (_patternSequence[i] == ZeroOrMany)
1826                 {
1827                     pattern.Add(patternBlock);
1828                     patternBlock = new PatternBlock
1829                     {
1830                         Type = PatternBlockType.Gap,
1831                         Start = 0,
1832                         Stop = long.MaxValue
1833                     };
1834                 }
1835                 else
1836                 {
1837                     patternBlock.Stop = i;
1838                 }
1839             }
1840             else // patternBlock.Type == PatternBlockType.Gap

```

```

1841         {
1842             if (_patternSequence[i] == _sequences.Constants.Any)
1843             {
1844                 patternBlock.Start++;
1845                 if (patternBlock.Stop < patternBlock.Start)
1846                 {
1847                     patternBlock.Stop = patternBlock.Start;
1848                 }
1849             }
1850             else if (_patternSequence[i] == ZeroOrMany)
1851             {
1852                 patternBlock.Stop = long.MaxValue;
1853             }
1854             else
1855             {
1856                 pattern.Add(patternBlock);
1857                 patternBlock = new PatternBlock
1858                 {
1859                     Type = PatternBlockType.Elements,
1860                     Start = i,
1861                     Stop = i
1862                 };
1863             }
1864         }
1865     }
1866     if (patternBlock.Type != PatternBlockType.Undefined)
1867     {
1868         pattern.Add(patternBlock);
1869     }
1870     return pattern;
1871 }
1872
1873 // match: search for regexp anywhere in text
1874 //int match(char* regexp, char* text)
1875 //{
1876 //    do
1877 //    {
1878 //    } while (*text++ != '\0');
1879 //    return 0;
1880 //}
1881
1882 // matchhere: search for regexp at beginning of text
1883 //int matchhere(char* regexp, char* text)
1884 //{
1885 //    if (regexp[0] == '\0')
1886 //        return 1;
1887 //    if (regexp[1] == '*')
1888 //        return matchstar(regexp[0], regexp + 2, text);
1889 //    if (regexp[0] == '$' && regexp[1] == '\0')
1890 //        return *text == '\0';
1891 //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1892 //        return matchhere(regexp + 1, text + 1);
1893 //    return 0;
1894 //}
1895
1896 // matchstar: search for c*regexp at beginning of text
1897 //int matchstar(int c, char* regexp, char* text)
1898 //{
1899 //    do
1900 //    {
1901 //        /* a * matches zero or more instances */
1902 //        if (matchhere(regexp, text))
1903 //            return 1;
1904 //    } while (*text != '\0' && (*text++ == c || c == '.'));
1905 //    return 0;
1906 //}
1907
1908 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
1909 //    long maximumGap)
1910 //{
1911 //    mininumGap = 0;
1912 //    maximumGap = 0;
1913 //    element = 0;
1914 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1915 //    {
1916 //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1917 //            mininumGap++;
1918 //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1919 //            maximumGap = long.MaxValue;
1920 //        else

```



```

1919         //          break;
1920     //    }
1921
1922     //    if (maximumGap < mininumGap)
1923     //        maximumGap = mininumGap;
1924     //}
1925
1926     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1927     private bool PatternMatchCore(LinkIndex element)
1928     {
1929         if (_patternPosition >= _pattern.Count)
1930         {
1931             _patternPosition = -2;
1932             return false;
1933         }
1934         var currentPatternBlock = _pattern[_patternPosition];
1935         if (currentPatternBlock.Type == PatternBlockType.Gap)
1936         {
1937             //var currentMatchingBlockLength = (_sequencePosition -
1938             ↪ _lastMatchedBlockPosition);
1939             if (_sequencePosition < currentPatternBlock.Start)
1940             {
1941                 _sequencePosition++;
1942                 return true; // Двигаемся дальше
1943             }
1944             // Это последний блок
1945             if (_pattern.Count == _patternPosition + 1)
1946             {
1947                 _patternPosition++;
1948                 _sequencePosition = 0;
1949                 return false; // Полное соответствие
1950             }
1951             else
1952             {
1953                 if (_sequencePosition > currentPatternBlock.Stop)
1954                 {
1955                     return false; // Соответствие невозможно
1956                 }
1957                 var nextPatternBlock = _pattern[_patternPosition + 1];
1958                 if (_patternSequence[nextPatternBlock.Start] == element)
1959                 {
1960                     if (nextPatternBlock.Start < nextPatternBlock.Stop)
1961                     {
1962                         _patternPosition++;
1963                         _sequencePosition = 1;
1964                     }
1965                     else
1966                     {
1967                         _patternPosition += 2;
1968                         _sequencePosition = 0;
1969                     }
1970                 }
1971             }
1972         }
1973         else // currentPatternBlock.Type == PatternBlockType.Elements
1974         {
1975             var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1976             if (_patternSequence[patternElementPosition] != element)
1977             {
1978                 return false; // Соответствие невозможно
1979             }
1980             if (patternElementPosition == currentPatternBlock.Stop)
1981             {
1982                 _patternPosition++;
1983                 _sequencePosition = 0;
1984             }
1985             else
1986             {
1987                 _sequencePosition++;
1988             }
1989         }
1990         return true;
1991         //if (_patternSequence[_patternPosition] != element)
1992         //    return false;
1993         //else
1994         //{
1995             //    _sequencePosition++;
1996             //    _patternPosition++;
1997             //    return true;
1998         //}

```

```

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 }

```

1.94 ./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     /// х*у - 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.

```

```

30  ///
31  /// Рост последовательности слева и справа.
32  /// Поиск со звёздочкой.
33  /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
34  /// так же проблема может быть решена при реализации дистанционных триггеров.
35  /// Нужны ли уникальные указатели вообще?
36  /// Что если обращение к информации будет происходить через содержимое всегда?
37  ///
38  /// Писать тесты.
39  ///
40  ///
41  /// Можно убрать зависимость от конкретной реализации Links,
42  /// на зависимость от абстрактного элемента, который может быть представлен несколькими
43  /// ↳ способами.
44  ///
45  /// Можно ли как-то сделать один общий интерфейс
46  ///
47  /// Блокчейн и/или гит для распределённой записи транзакций.
48  ///
49  /// </remarks>
50  public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
51  {
52  /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
53  /// ↳ связей.</summary>
54  public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
55
56  public SequencesOptions<LinkIndex> Options { get; }
57  public SynchronizedLinks<LinkIndex> Links { get; }
58  private readonly ISynchronization _sync;
59
60  public LinksConstants<LinkIndex> Constants { get; }
61
62  [MethodImpl(MethodImplOptions.AggressiveInlining)]
63  public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
64  {
65      Links = links;
66      _sync = links.SyncRoot;
67      Options = options;
68      Options.ValidateOptions();
69      Options.InitOptions(Links);
70      Constants = links.Constants;
71  }
72
73  [MethodImpl(MethodImplOptions.AggressiveInlining)]
74  public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
75  ↳ SequencesOptions<LinkIndex>()) { }
76
77  [MethodImpl(MethodImplOptions.AggressiveInlining)]
78  public bool IsSequence(LinkIndex sequence)
79  {
80      return _sync.ExecuteReadOperation(() =>
81      {
82          if (Options.UseSequenceMarker)
83          {
84              return Options.MarkedSequenceMatcher.IsMatched(sequence);
85          }
86          return !Links.Unsync.IsPartialPoint(sequence);
87      });
88  }
89
90  [MethodImpl(MethodImplOptions.AggressiveInlining)]
91  private LinkIndex GetSequenceByElements(LinkIndex sequence)
92  {
93      if (Options.UseSequenceMarker)
94      {
95          return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
96      }
97      return sequence;
98  }
99
100  [MethodImpl(MethodImplOptions.AggressiveInlining)]
101  private LinkIndex GetSequenceElements(LinkIndex sequence)
102  {
103      if (Options.UseSequenceMarker)
104      {
105          var linkContents = new Link<ulong>(Links.GetLink(sequence));

```

```

104         if (linkContents.Source == Options.SequenceMarkerLink)
105         {
106             return linkContents.Target;
107         }
108         if (linkContents.Target == Options.SequenceMarkerLink)
109         {
110             return linkContents.Source;
111         }
112     }
113     return sequence;
114 }
115
116 #region Count
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 public LinkIndex Count(ICollection<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) -
166                     ↪ 1;
167             }
168             return Links.Count(any, elementsLink);
169         }
170         return Links.Count(any, restrictions[0]);
171     }
172     throw new NotImplementedException();
173 }
174
175 #endregion
176
177 #region Create
178
179 [MethodImpl(MethodImplOptions.AggressiveInlining)]
180 public LinkIndex Create(ICollection<LinkIndex> restrictions)
181 {
182     return _sync.ExecuteWriteOperation(() =>

```

```

182     {
183         if (restrictions.IsNullOrEmpty())
184         {
185             return Constants.Null;
186         }
187         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
188         return CreateCore(restrictions);
189     });
190 }
191
192 [MethodImpl(MethodImplOptions.AggressiveInlining)]
193 private LinkIndex CreateCore(ICollection<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(ICollection<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<ICollection<LinkIndex>, LinkIndex> handler, ICollection<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,
256                         ↪ Options.SequenceMarkerLink, any));
257                 }
258                 else

```

```

258         {
259             return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
260                 ↪ any));
261         }
262     if (Options.UseSequenceMarker)
263     {
264         var sequenceLinkValues = Links.Unsync.GetLink(link);
265         if (sequenceLinkValues[Constants.SourcePart] ==
266             ↪ Options.SequenceMarkerLink)
267         {
268             link = sequenceLinkValues[Constants.TargetPart];
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>
296     ↪ values)
297 {
298     var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
299     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
300     ↪ Id.
301     Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
302     ↪ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
303     ↪ matcher.HandleFullMatched;
304     //if (sequence.Length >= 2)
305     if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
306     {
307         return Constants.Break;
308     }
309     var last = values.Count - 2;
310     for (var i = 1; i < last; i++)
311     {
312         if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
313             ↪ Constants.Continue)
314         {
315             return Constants.Break;
316         }
317     }
318     if (values.Count >= 3)
319     {
320         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
321             ↪ != Constants.Continue)
322         {
323             return Constants.Break;
324         }
325     }
326     return Constants.Continue;
327 }
328
329 [MethodImpl(MethodImplOptions.AggressiveInlining)]
330 private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
331     ↪ left, LinkIndex right)
332 {
333     return Links.Unsync.Each(doublet =>

```

```

327     {
328         var doubletIndex = doublet[Constants.IndexPart];
329         if (StepRight(handler, doubletIndex, right) != Constants.Continue)
330         {
331             return Constants.Break;
332         }
333         if (left != doubletIndex)
334         {
335             return PartialStepRight(handler, doubletIndex, right);
336         }
337         return Constants.Continue;
338     }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
339 }
340
341 [MethodImpl(MethodImplOptions.AggressiveInlining)]
342 private LinkIndex StepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↳ LinkIndex right) => Links.Unsync.Each(rightStep => TryStepRightUp(handler, right,
    ↳ rightStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, left,
    ↳ Constants.Any));
343
344 [MethodImpl(MethodImplOptions.AggressiveInlining)]
345 private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ right, LinkIndex stepFrom)
346 {
347     var upStep = stepFrom;
348     var firstSource = Links.Unsync.GetTarget(upStep);
349     while (firstSource != right && firstSource != upStep)
350     {
351         upStep = firstSource;
352         firstSource = Links.Unsync.GetSource(upStep);
353     }
354     if (firstSource == right)
355     {
356         return handler(new LinkAddress<LinkIndex>(stepFrom));
357     }
358     return Constants.Continue;
359 }
360
361 [MethodImpl(MethodImplOptions.AggressiveInlining)]
362 private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↳ LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
    ↳ leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
    ↳ right));
363
364 [MethodImpl(MethodImplOptions.AggressiveInlining)]
365 private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ left, LinkIndex stepFrom)
366 {
367     var upStep = stepFrom;
368     var firstTarget = Links.Unsync.GetSource(upStep);
369     while (firstTarget != left && firstTarget != upStep)
370     {
371         upStep = firstTarget;
372         firstTarget = Links.Unsync.GetTarget(upStep);
373     }
374     if (firstTarget == left)
375     {
376         return handler(new LinkAddress<LinkIndex>(stepFrom));
377     }
378     return Constants.Continue;
379 }
380
381 #endregion
382
383 #region Update
384
385 [MethodImpl(MethodImplOptions.AggressiveInlining)]
386 public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
387 {
388     var sequence = restrictions.SkipFirst();
389     var newSequence = substitution.SkipFirst();
390     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
391     {
392         return Constants.Null;
393     }
394     if (sequence.IsNullOrEmpty())
395     {
396         return Create(substitution);
397     }

```

```

398     if (newSequence.IsNullOrEmpty())
399     {
400         Delete(restrictions);
401         return Constants.Null;
402     }
403     return _sync.ExecuteWriteOperation((Func<ulong>)(() =>
404     {
405         ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
406         Links.EnsureLinkExists(newSequence);
407         return UpdateCore(sequence, newSequence);
408     }));
409 }
410
411 [MethodImpl(MethodImplOptions.AggressiveInlining)]
412 private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence)
413 {
414     LinkIndex bestVariant;
415     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
416         ↪ !sequence.EqualTo(newSequence))
417     {
418         bestVariant = CompactCore(newSequence);
419     }
420     else
421     {
422         bestVariant = CreateCore(newSequence);
423     }
424     // TODO: Check all options only ones before loop execution
425     // Возможно нужно две версии Each, возвращающий фактические последовательности и с
426     ↪ маркером,
427     // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
428     ↪ можно получить имея только фактические последовательности.
429     foreach (var variant in Each(sequence))
430     {
431         if (variant != bestVariant)
432         {
433             UpdateOneCore(variant, bestVariant);
434         }
435     }
436     return bestVariant;
437 }
438
439 [MethodImpl(MethodImplOptions.AggressiveInlining)]
440 private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
441 {
442     if (Options.UseGarbageCollection)
443     {
444         var sequenceElements = GetSequenceElements(sequence);
445         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
446         var sequenceLink = GetSequenceByElements(sequenceElements);
447         var newSequenceElements = GetSequenceElements(newSequence);
448         var newSequenceLink = GetSequenceByElements(newSequenceElements);
449         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
450         {
451             if (sequenceLink != Constants.Null)
452             {
453                 Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
454             }
455             Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
456         }
457         ClearGarbage(sequenceElementsContents.Source);
458         ClearGarbage(sequenceElementsContents.Target);
459     }
460     else
461     {
462         if (Options.UseSequenceMarker)
463         {
464             var sequenceElements = GetSequenceElements(sequence);
465             var sequenceLink = GetSequenceByElements(sequenceElements);
466             var newSequenceElements = GetSequenceElements(newSequence);
467             var newSequenceLink = GetSequenceByElements(newSequenceElements);
468             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
469             {
470                 if (sequenceLink != Constants.Null)
471                 {
472                     Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
473                 }
474                 Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
475             }
476         }
477     }
478 }

```



```

473     }
474     else
475     {
476         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
477         {
478             Links.Unsync.MergeAndDelete(sequence, newSequence);
479         }
480     }
481 }
482 }
483
484 #endregion
485
486 #region Delete
487
488 [MethodImpl(MethodImplOptions.AggressiveInlining)]
489 public void Delete(ICollection<LinkIndex> restrictions)
490 {
491     _sync.ExecuteWriteOperation(() =>
492     {
493         var sequence = restrictions.SkipFirst();
494         // TODO: Check all options only ones before loop execution
495         foreach (var linkToDelete in Each(sequence))
496         {
497             DeleteOneCore(linkToDelete);
498         }
499     });
500 }
501
502 [MethodImpl(MethodImplOptions.AggressiveInlining)]
503 private void DeleteOneCore(LinkIndex link)
504 {
505     if (Options.UseGarbageCollection)
506     {
507         var sequenceElements = GetSequenceElements(link);
508         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
509         var sequenceLink = GetSequenceByElements(sequenceElements);
510         if (Options.UseCascadeDelete || CountUsages(link) == 0)
511         {
512             if (sequenceLink != Constants.Null)
513             {
514                 Links.Unsync.Delete(sequenceLink);
515             }
516             Links.Unsync.Delete(link);
517         }
518         ClearGarbage(sequenceElementsContents.Source);
519         ClearGarbage(sequenceElementsContents.Target);
520     }
521     else
522     {
523         if (Options.UseSequenceMarker)
524         {
525             var sequenceElements = GetSequenceElements(link);
526             var sequenceLink = GetSequenceByElements(sequenceElements);
527             if (Options.UseCascadeDelete || CountUsages(link) == 0)
528             {
529                 if (sequenceLink != Constants.Null)
530                 {
531                     Links.Unsync.Delete(sequenceLink);
532                 }
533                 Links.Unsync.Delete(link);
534             }
535         }
536         else
537         {
538             if (Options.UseCascadeDelete || CountUsages(link) == 0)
539             {
540                 Links.Unsync.Delete(link);
541             }
542         }
543     }
544 }
545
546 #endregion
547
548 #region Compactification
549
550 [MethodImpl(MethodImplOptions.AggressiveInlining)]
551 public void CompactAll()

```

```

552 {
553     _sync.ExecuteWriteOperation(() =>
554     {
555         var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
556         for (int i = 0; i < sequences.Count; i++)
557         {
558             var sequence = this.ToList(sequences[i]);
559             Compact(sequence.ShiftRight());
560         }
561     });
562 }
563
564 /// <remarks>
565 /// bestVariant можно выбирать по максимальному числу использований,
566 /// но балансированный позволяет гарантировать уникальность (если есть возможность,
567 /// гарантировать его использование в других местах).
568 ///
569 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
570 /// </remarks>
571 [MethodImpl(MethodImplOptions.AggressiveInlining)]
572 public LinkIndex Compact(ICollection<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(ICollection<LinkIndex> sequence) => UpdateCore(sequence,
587     ↪ sequence);
588
589 #endregion
590
591 #region Garbage Collection
592
593 /// <remarks>
594 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
595 ↪ определить извне или в унаследованном классе
596 /// </remarks>
597 [MethodImpl(MethodImplOptions.AggressiveInlining)]
598 private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
599     ↪ !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
600
601 [MethodImpl(MethodImplOptions.AggressiveInlining)]
602 private void ClearGarbage(LinkIndex link)
603 {
604     if (IsGarbage(link))
605     {
606         var contents = new Link<ulong>(Links.GetLink(link));
607         Links.Unsync.Delete(link);
608         ClearGarbage(contents.Source);
609         ClearGarbage(contents.Target);
610     }
611 }
612
613 #endregion
614
615 #region Walkers
616
617 [MethodImpl(MethodImplOptions.AggressiveInlining)]
618 public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
619 {
620     return _sync.ExecuteReadOperation(() =>
621     {
622         var links = Links.Unsync;
623         foreach (var part in Options.Walker.Walk(sequence))
624         {
625             if (!handler(part))
626             {
627                 return false;

```

```

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

```

```

703     {
704         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
705     }
706     return _links.Constants.Continue;
707 }
708
709 [MethodImpl(MethodImplOptions.AggressiveInlining)]
710 public LinkIndex HandleFullMatchedSequence(ICollection<LinkIndex> restrictions)
711 {
712     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
713     var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
714     if (sequence != _links.Constants.Null && FullMatch(sequenceToMatch) &&
715         ↪ _results.Add(sequenceToMatch))
716     {
717         return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
718     }
719     return _links.Constants.Continue;
720 }
721
722 /// <remarks>
723 /// TODO: Add support for LinksConstants.Any
724 /// </remarks>
725 [MethodImpl(MethodImplOptions.AggressiveInlining)]
726 public bool PartialMatch(LinkIndex sequenceToMatch)
727 {
728     _filterPosition = -1;
729     foreach (var part in Walk(sequenceToMatch))
730     {
731         if (!PartialMatchCore(part))
732         {
733             break;
734         }
735     }
736     return _filterPosition == _patternSequence.Count - 1;
737 }
738
739 [MethodImpl(MethodImplOptions.AggressiveInlining)]
740 private bool PartialMatchCore(LinkIndex element)
741 {
742     if (_filterPosition == (_patternSequence.Count - 1))
743     {
744         return false; // Нашлось
745     }
746     if (_filterPosition >= 0)
747     {
748         if (element == _patternSequence[_filterPosition + 1])
749         {
750             _filterPosition++;
751         }
752         else
753         {
754             _filterPosition = -1;
755         }
756     }
757     if (_filterPosition < 0)
758     {
759         if (element == _patternSequence[0])
760         {
761             _filterPosition = 0;
762         }
763     }
764     return true; // Ищем дальше
765 }
766
767 [MethodImpl(MethodImplOptions.AggressiveInlining)]
768 public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
769 {
770     if (PartialMatch(sequenceToMatch))
771     {
772         _results.Add(sequenceToMatch);
773     }
774 }
775
776 [MethodImpl(MethodImplOptions.AggressiveInlining)]
777 public LinkIndex HandlePartialMatched(ICollection<LinkIndex> restrictions)
778 {
779     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
780     if (PartialMatch(sequenceToMatch))
781     {

```

```

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>
800 ↪ sequencesToMatch)
801 {
802     foreach (var sequenceToMatch in sequencesToMatch)
803     {
804         if (PartialMatch(sequenceToMatch))
805         {
806             _readAsElements.Add(sequenceToMatch);
807             _results.Add(sequenceToMatch);
808         }
809     }
810 }
811
812 #endregion
813 }
814 }

```

1.95 ./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[]>
13         ↪ groupedSequence)
14         {
15             var finalSequence = new TLink[groupedSequence.Count];
16             for (var i = 0; i < finalSequence.Length; i++)
17             {
18                 var part = groupedSequence[i];
19                 finalSequence[i] = part.Length == 1 ? part[0] :
20                 ↪ sequences.Create(part.ShiftRight());
21             }
22             return sequences.Create(finalSequence.ShiftRight());
23         }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
27         {
28             var list = new List<TLink>();
29             var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
30             sequences.Each(filler.AddSkipFirstAndReturnConstant, new
31             ↪ LinkAddress<TLink>(sequence));
32             return list;
33         }
34     }
35 }

```

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

```

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

```

```

5 using Platform.Converters;
6 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
7 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
8 using Platform.Data.Doublets.Sequences.Converters;
9 using Platform.Data.Doublets.Sequences.Walkers;
10 using Platform.Data.Doublets.Sequences.Indexes;
11 using Platform.Data.Doublets.Sequences.CriterionMatchers;
12 using System.Runtime.CompilerServices;
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 SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
        ↳ ILinks<TLink> must contain GetConstants function.
19     {
20         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
21
22         public TLink SequenceMarkerLink
23         {
24             [MethodImpl(MethodImplOptions.AggressiveInlining)]
25             get;
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             set;
28         }
29
30         public bool UseCascadeUpdate
31         {
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             get;
34             [MethodImpl(MethodImplOptions.AggressiveInlining)]
35             set;
36         }
37
38         public bool UseCascadeDelete
39         {
40             [MethodImpl(MethodImplOptions.AggressiveInlining)]
41             get;
42             [MethodImpl(MethodImplOptions.AggressiveInlining)]
43             set;
44         }
45
46         public bool UseIndex
47         {
48             [MethodImpl(MethodImplOptions.AggressiveInlining)]
49             get;
50             [MethodImpl(MethodImplOptions.AggressiveInlining)]
51             set;
52         } // TODO: Update Index on sequence update/delete.
53
54         public bool UseSequenceMarker
55         {
56             [MethodImpl(MethodImplOptions.AggressiveInlining)]
57             get;
58             [MethodImpl(MethodImplOptions.AggressiveInlining)]
59             set;
60         }
61
62         public bool UseCompression
63         {
64             [MethodImpl(MethodImplOptions.AggressiveInlining)]
65             get;
66             [MethodImpl(MethodImplOptions.AggressiveInlining)]
67             set;
68         }
69
70         public bool UseGarbageCollection
71         {
72             [MethodImpl(MethodImplOptions.AggressiveInlining)]
73             get;
74             [MethodImpl(MethodImplOptions.AggressiveInlining)]
75             set;
76         }
77
78         public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting
79         {
80             [MethodImpl(MethodImplOptions.AggressiveInlining)]
81             get;
82             [MethodImpl(MethodImplOptions.AggressiveInlining)]
83             set;

```

```

84     }
85
86     public bool EnforceSingleSequenceVersionOnWriteBasedOnNew
87     {
88         [MethodImpl(MethodImplOptions.AggressiveInlining)]
89         get;
90         [MethodImpl(MethodImplOptions.AggressiveInlining)]
91         set;
92     }
93
94     public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher
95     {
96         [MethodImpl(MethodImplOptions.AggressiveInlining)]
97         get;
98         [MethodImpl(MethodImplOptions.AggressiveInlining)]
99         set;
100     }
101
102     public IConverter<IList<TLink>, TLink> LinksToSequenceConverter
103     {
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         get;
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         set;
108     }
109
110     public ISequenceIndex<TLink> Index
111     {
112         [MethodImpl(MethodImplOptions.AggressiveInlining)]
113         get;
114         [MethodImpl(MethodImplOptions.AggressiveInlining)]
115         set;
116     }
117
118     public ISequenceWalker<TLink> Walker
119     {
120         [MethodImpl(MethodImplOptions.AggressiveInlining)]
121         get;
122         [MethodImpl(MethodImplOptions.AggressiveInlining)]
123         set;
124     }
125
126     public bool ReadFullSequence
127     {
128         [MethodImpl(MethodImplOptions.AggressiveInlining)]
129         get;
130         [MethodImpl(MethodImplOptions.AggressiveInlining)]
131         set;
132     }
133
134     // TODO: Реализовать компактификацию при чтении
135     //public bool EnforceSingleSequenceVersionOnRead { get; set; }
136     //public bool UseRequestMarker { get; set; }
137     //public bool StoreRequestResults { get; set; }
138
139     [MethodImpl(MethodImplOptions.AggressiveInlining)]
140     public void InitOptions(ISynchronizedLinks<TLink> links)
141     {
142         if (UseSequenceMarker)
143         {
144             if (!_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
145             {
146                 SequenceMarkerLink = links.CreatePoint();
147             }
148             else
149             {
150                 if (!links.Exists(SequenceMarkerLink))
151                 {
152                     var link = links.CreatePoint();
153                     if (!_equalityComparer.Equals(link, SequenceMarkerLink))
154                     {
155                         throw new InvalidOperationException("Cannot recreate sequence marker
156                             ↪ link.");
157                     }
158                 }
159             }
160             if (MarkedSequenceMatcher == null)
161             {
162                 MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
163                     ↪ 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
173                 ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
174                 ↪ MarkedSequenceMatcher);
175         }
176         else
177         {
178             totalSequenceSymbolFrequencyCounter = new
179                 ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
180         }
181         var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
182             ↪ totalSequenceSymbolFrequencyCounter);
183         var compressingConverter = new CompressingConverter<TLink>(links,
184             ↪ balancedVariantConverter, doubletFrequenciesCache);
185         LinksToSequenceConverter = compressingConverter;
186     }
187 }
188 else
189 {
190     if (LinksToSequenceConverter == null)
191     {
192         LinksToSequenceConverter = balancedVariantConverter;
193     }
194 }
195 if (UseIndex && Index == null)
196 {
197     Index = new SequenceIndex<TLink>(links);
198 }
199 if (Walker == null)
200 {
201     Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
202 }
203 }
204
205 [MethodImpl(MethodImplOptions.AggressiveInlining)]
206 public void ValidateOptions()
207 {
208     if (UseGarbageCollection && !UseSequenceMarker)
209     {
210         throw new NotSupportedException("To use garbage collection UseSequenceMarker
211             ↪ option must be on.");
212     }
213 }
214 }
215 }

```

1.97 ./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 }

```

1.98 ./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 }

```

1.99 ./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;

```

```

40         do
41         {
42             length *= 2;
43 #if USEARRAYPOOL
44             var nextArray = ArrayPool.Allocate<ulong>(length);
45 #else
46             var nextArray = new TLink[length];
47 #endif
48             hasElements = false;
49             for (var i = 0; i < array.Length; i++)
50             {
51                 var candidate = array[i];
52                 if (!_equalityComparer.Equals(array[i], default))
53                 {
54                     continue;
55                 }
56                 var doubletOffset = i * 2;
57                 if (!_isElement(candidate))
58                 {
59                     nextArray[doubletOffset] = candidate;
60                 }
61                 else
62                 {
63                     var links = _links;
64                     var link = links.GetLink(candidate);
65                     var linkSource = links.GetSource(link);
66                     var linkTarget = links.GetTarget(link);
67                     nextArray[doubletOffset] = linkSource;
68                     nextArray[doubletOffset + 1] = linkTarget;
69                     if (!hasElements)
70                     {
71                         hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
72                     }
73                 }
74             }
75 #if USEARRAYPOOL
76             if (array.Length > 1)
77             {
78                 ArrayPool.Free(array);
79             }
80 #endif
81             array = nextArray;
82         }
83         while (hasElements);
84         var filledElementsCount = CountFilledElements(array);
85         if (filledElementsCount == array.Length)
86         {
87             return array;
88         }
89         else
90         {
91             return CopyFilledElements(array, filledElementsCount);
92         }
93     }
94
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
97     {
98         var finalArray = new TLink[filledElementsCount];
99         for (int i = 0, j = 0; i < array.Length; i++)
100         {
101             if (!_equalityComparer.Equals(array[i], default))
102             {
103                 finalArray[j] = array[i];
104                 j++;
105             }
106         }
107 #if USEARRAYPOOL
108         ArrayPool.Free(array);
109 #endif
110         return finalArray;
111     }
112
113     [MethodImpl(MethodImplOptions.AggressiveInlining)]
114     private static int CountFilledElements(TLink[] array)
115     {
116         var count = 0;
117         for (var i = 0; i < array.Length; i++)
118         {

```

```

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

```

1.100 ./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>
14             ↪ isElement) : base(links, stack, isElement) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
18             ↪ stack, links.IsPartialPoint) { }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override TLink GetNextElementAfterPop(TLink element) =>
22             ↪ _links.GetTarget(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override TLink GetNextElementAfterPush(TLink element) =>
26             ↪ _links.GetSource(element);
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override IEnumerable<TLink> WalkContents(TLink element)
30         {
31             var parts = _links.GetLink(element);
32             for (var i = _links.Constants.SourcePart; i < parts.Count; i++)
33             {
34                 var part = parts[i];
35                 if (IsElement(part))
36                 {
37                     yield return part;
38                 }
39             }
40         }
41     }
42 }

```

1.101 ./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>,
11         ↪ ISequenceWalker<TLink>
12     {
13         private readonly IStack<TLink> _stack;
14         private readonly Func<TLink, bool> _isElement;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
18             ↪ isElement) : base(links)
19         {
20             _stack = stack;
21             _isElement = isElement;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

23     protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
24         ↪ stack, links.IsPartialPoint) { }
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     public IEnumerable<TLink> Walk(TLink sequence)
28     {
29         _stack.Clear();
30         var element = sequence;
31         if (IsElement(element))
32         {
33             yield return element;
34         }
35         else
36         {
37             while (true)
38             {
39                 if (IsElement(element))
40                 {
41                     if (_stack.IsEmpty)
42                     {
43                         break;
44                     }
45                     element = _stack.Pop();
46                     foreach (var output in WalkContents(element))
47                     {
48                         yield return output;
49                     }
50                     element = GetNextElementAfterPop(element);
51                 }
52                 else
53                 {
54                     _stack.Push(element);
55                     element = GetNextElementAfterPush(element);
56                 }
57             }
58         }
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);
62
63         [MethodImpl(MethodImplOptions.AggressiveInlining)]
64         protected abstract TLink GetNextElementAfterPop(TLink element);
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         protected abstract TLink GetNextElementAfterPush(TLink element);
68
69         [MethodImpl(MethodImplOptions.AggressiveInlining)]
70         protected abstract IEnumerable<TLink> WalkContents(TLink element);
71     }
72 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Collections.Stacks;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Stacks
8  {
9      public class Stack<TLink> : LinksOperatorBase<TLink>, IStack<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13
14         private readonly TLink _stack;
15
16         public bool IsEmpty
17         {
18             [MethodImpl(MethodImplOptions.AggressiveInlining)]
19             get => _equalityComparer.Equals(Peek(), _stack);
20         }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public Stack(ILinks<TLink> links, TLink stack) : base(links) => _stack = stack;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         private TLink GetStackMarker() => _links.GetSource(_stack);
27     }

```

```

27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     private TLink GetTop() => _links.GetTarget(_stack);
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     public TLink Peek() => _links.GetTarget(GetTop());
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     public TLink Pop()
35     {
36         var element = Peek();
37         if (!_equalityComparer.Equals(element, _stack))
38         {
39             var top = GetTop();
40             var previousTop = _links.GetSource(top);
41             _links.Update(_stack, GetStackMarker(), previousTop);
42             _links.Delete(top);
43         }
44         return element;
45     }
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
49     ↪ _links.GetOrCreate(GetTop(), element));
50 }

```

1.103 ./csharp/Platform.Data.Doublets/Stacks/StackExtensions.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.Stacks
6 {
7     public static class StackExtensions
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public static TLink CreateStack<TLink>(this ILinks<TLink> links, TLink stackMarker)
11        {
12            var stackPoint = links.CreatePoint();
13            var stack = links.Update(stackPoint, stackMarker, stackPoint);
14            return stack;
15        }
16    }
17 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Data.Doublets;
5 using Platform.Threading.Synchronization;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets
10 {
11     /// <remarks>
12     /// TODO: Autogeneration of synchronized wrapper (decorator).
13     /// TODO: Try to unfold code of each method using IL generation for performance improvements.
14     /// TODO: Or even to unfold multiple layers of implementations.
15     /// </remarks>
16     public class SynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress>
17     {
18         public LinksConstants<TLinkAddress> Constants
19         {
20             [MethodImpl(MethodImplOptions.AggressiveInlining)]
21             get;
22         }
23
24         public ISynchronization SyncRoot
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28         }
29
30         public ILinks<TLinkAddress> Sync
31         {
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             get;
34         }
35     }
36 }

```

```

35
36 public ILinks<TLinkAddress> Unsync
37 {
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     get;
40 }
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
    ↳ ReaderWriterLockSynchronization(), links) { }
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
47 {
48     SyncRoot = synchronization;
49     Sync = this;
50     Unsync = links;
51     Constants = links.Constants;
52 }
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 public TLinkAddress Count(IList<TLinkAddress> restriction) =>
    ↳ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
    ↳ IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
    ↳ restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
    ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
    ↳ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
    ↳ Unsync.Update);
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 public void Delete(IList<TLinkAddress> restrictions) =>
    ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
68
69 //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
    ↳ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
70 //{
71 //    if (restriction != null && substitution != null &&
    ↳ !substitution.EqualTo(restriction))
72 //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
    ↳ substitution, substitutedHandler, Unsync.Trigger);
73
74 //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
    ↳ substitutedHandler, Unsync.Trigger);
75 //}
76 }
77 }

```

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

```

1 using System;
2 using System.Text;
3 using System.Collections.Generic;
4 using System.Runtime.CompilerServices;
5 using Platform.Singletons;
6 using Platform.Data.Doublets.Unicode;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets
11 {
12     public static class UInt64LinksExtensions
13     {
14         public static readonly LinksConstants<ulong> Constants =
            ↳ Default<LinksConstants<ulong>>.Instance;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
21         {

```

```

22     if (sequence == null)
23     {
24         return false;
25     }
26     var constants = links.Constants;
27     for (var i = 0; i < sequence.Length; i++)
28     {
29         if (sequence[i] == constants.Any)
30         {
31             return true;
32         }
33     }
34     return false;
35 }
36
37 [MethodImpl(MethodImplOptions.AggressiveInlining)]
38 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
  ↳ Func<Link<ulong>, bool> isElement, bool renderIndex = false, bool renderDebug =
  ↳ false)
39 {
40     var sb = new StringBuilder();
41     var visited = new HashSet<ulong>();
42     links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
  ↳ innerSb.Append(link.Index), renderIndex, renderDebug);
43     return sb.ToString();
44 }
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
  ↳ Func<Link<ulong>, bool> isElement, Action<StringBuilder, Link<ulong>> appendElement,
  ↳ bool renderIndex = false, bool renderDebug = false)
48 {
49     var sb = new StringBuilder();
50     var visited = new HashSet<ulong>();
51     links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
  ↳ renderDebug);
52     return sb.ToString();
53 }
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
  ↳ HashSet<ulong> visited, ulong linkIndex, Func<Link<ulong>, bool> isElement,
  ↳ Action<StringBuilder, Link<ulong>> appendElement, bool renderIndex = false, bool
  ↳ renderDebug = false)
57 {
58     if (sb == null)
59     {
60         throw new ArgumentNullException(nameof(sb));
61     }
62     if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
  ↳ Constants.Itself)
63     {
64         return;
65     }
66     if (links.Exists(linkIndex))
67     {
68         if (visited.Add(linkIndex))
69         {
70             sb.Append('(');
71             var link = new Link<ulong>(links.GetLink(linkIndex));
72             if (renderIndex)
73             {
74                 sb.Append(link.Index);
75                 sb.Append(':');
76             }
77             if (link.Source == link.Index)
78             {
79                 sb.Append(link.Index);
80             }
81             else
82             {
83                 var source = new Link<ulong>(links.GetLink(link.Source));
84                 if (isElement(source))
85                 {
86                     appendElement(sb, source);
87                 }
88                 else
89                 {

```

```

90         links.AppendStructure(sb, visited, source.Index, isElement,
91                               ↪ appendElement, renderIndex);
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,
108                                   ↪ appendElement, renderIndex);
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 }

```

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

```

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         /// {

```



```

34     ///     public ulong TransactionId;
35     ///     public UniqueTimestamp Timestamp;
36     ///     public TransactionItemType Type;
37     ///     public Link Source;
38     ///     public Link Linker;
39     ///     public Link Target;
40     /// }
41     ///
42     /// Или
43     ///
44     /// public struct TransitionHeader
45     /// {
46     ///     public ulong TransactionIdCombined;
47     ///     public ulong TimestampCombined;
48     ///
49     ///     public ulong TransactionId
50     ///     {
51     ///         get
52     ///         {
53     ///             return (ulong) mask & TransactionIdCombined;
54     ///         }
55     ///     }
56     ///
57     ///     public UniqueTimestamp Timestamp
58     ///     {
59     ///         get
60     ///         {
61     ///             return (UniqueTimestamp)mask & TransactionIdCombined;
62     ///         }
63     ///     }
64     ///
65     ///     public TransactionItemType Type
66     ///     {
67     ///         get
68     ///         {
69     ///             // Использовать по одному биту из TransactionId и Timestamp,
70     ///             // для значения в 2 бита, которое представляет тип операции
71     ///             throw new NotImplementedException();
72     ///         }
73     ///     }
74     /// }
75     ///
76     /// private struct Transition
77     /// {
78     ///     public TransitionHeader Header;
79     ///     public Link Source;
80     ///     public Link Linker;
81     ///     public Link Target;
82     /// }
83     ///
84     /// </remarks>
85     public struct Transition : IEquatable<Transition>
86     {
87         public static readonly long Size = Structure<Transition>.Size;
88
89         public readonly ulong TransactionId;
90         public readonly Link<ulong> Before;
91         public readonly Link<ulong> After;
92         public readonly Timestamp Timestamp;
93
94         [MethodImpl(MethodImplOptions.AggressiveInlining)]
95         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
96         ↪ transactionId, Link<ulong> before, Link<ulong> after)
97         {
98             TransactionId = transactionId;
99             Before = before;
100            After = after;
101            Timestamp = uniqueTimestampFactory.Create();
102        }
103
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
106         ↪ transactionId, Link<ulong> before) : this(uniqueTimestampFactory, transactionId,
107         ↪ before, default) { }
108
109         [MethodImpl(MethodImplOptions.AggressiveInlining)]
110         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
111         ↪ transactionId) : this(uniqueTimestampFactory, transactionId, default, default) {
112         ↪ }

```

```

108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
110     ↳ {After}";
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 public override bool Equals(object obj) => obj is Transition transition ?
114     ↳ Equals(transition) : false;
115
116 [MethodImpl(MethodImplOptions.AggressiveInlining)]
117 public override int GetHashCode() => (TransactionId, Before, After,
118     ↳ Timestamp).GetHashCode();
119
120 [MethodImpl(MethodImplOptions.AggressiveInlining)]
121 public bool Equals(Transition other) => TransactionId == other.TransactionId &&
122     ↳ Before == other.Before && After == other.After && Timestamp == other.Timestamp;
123
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 public static bool operator ==(Transition left, Transition right) =>
126     ↳ left.Equals(right);
127
128 [MethodImpl(MethodImplOptions.AggressiveInlining)]
129 public static bool operator !=(Transition left, Transition right) => !(left ==
130     ↳ right);
131
132 }
133
134 /// <remarks>
135 /// Другие варианты реализации транзакций (атомарности):
136 /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
137     ↳ Target)) и индексов.
138 /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
139     ↳ потребуется решить вопрос
140 /// со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
141     ↳ пересечениями идентификаторов.
142 ///
143 /// Где хранить промежуточный список транзакций?
144 ///
145 /// В оперативной памяти:
146 /// Минусы:
147 /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
148     ↳ так как нужно отдельно выделять память под список трансформаций.
149 /// 2. Выделенной оперативной памяти может не хватить, в том случае,
150     ↳ если транзакция использует слишком много трансформаций.
151 /// -> Можно использовать жёсткий диск для слишком длинных транзакций.
152 /// -> Максимальный размер списка трансформаций можно ограничить / задать
153     ↳ константой.
154 /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
155     ↳ создавая задержку.
156 ///
157 /// На жёстком диске:
158 /// Минусы:
159 /// 1. Длительный отклик, на запись каждой трансформации.
160 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
161     ↳ -> Это может решаться упаковкой/исключением дублирующих операций.
162     ↳ -> Также это может решаться тем, что короткие транзакции вообще
163     ↳ не будут записываться в случае отката.
164 /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
165     ↳ операции (трансформации)
166     ↳ будут записаны в лог.
167 ///
168 /// </remarks>
169 public class Transaction : DisposableBase
170 {
171     private readonly Queue<Transition> _transitions;
172     private readonly UInt64LinksTransactionsLayer _layer;
173     public bool IsCommitted { get; private set; }
174     public bool IsReverted { get; private set; }
175
176     [MethodImpl(MethodImplOptions.AggressiveInlining)]
177     public Transaction(UInt64LinksTransactionsLayer layer)
178     {
179         _layer = layer;
180         if (_layer._currentTransactionId != 0)
181         {
182             throw new NotSupportedException("Nested transactions not supported.");
183         }
184         IsCommitted = false;
185         IsReverted = false;
186     }
187 }

```

```

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._lastCommittedTransactionId = _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,
206 ↪ Transaction transaction)
207     {
208         layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
209         layer._currentTransactionTransitions = transaction._transitions;
210         layer._currentTransaction = transaction;
211     }
212
213     [MethodImpl(MethodImplOptions.AggressiveInlining)]
214     public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
215     {
216         if (transaction.IsReverted)
217         {
218             throw new InvalidOperationException("Transation is reverted.");
219         }
220         if (transaction.IsCommitted)
221         {
222             throw new InvalidOperationException("Transation is committed.");
223         }
224     }
225
226     [MethodImpl(MethodImplOptions.AggressiveInlining)]
227     protected override void Dispose(bool manual, bool wasDisposed)
228     {
229         if (!wasDisposed && _layer != null && !_layer.Disposable.IsDisposed)
230         {
231             if (!IsCommitted && !IsReverted)
232             {
233                 Revert();
234             }
235             _layer.ResetCurrentTransation();
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 _lastCommittedTransition;
247     private ulong _currentTransactionId;
248     private Queue<Transition> _currentTransactionTransitions;
249     private Transaction _currentTransaction;
250     private ulong _lastCommittedTransactionId;
251
252     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

253 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
254     : base(links)
255 {
256     if (string.IsNullOrEmpty(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 lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
265     var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
266     if (!lastCommittedTransition.Equals(lastWrittenTransition))
267     {
268         Dispose();
269         throw new NotSupportedException("Database is damaged, autorecovery is not
270             ↳ supported yet.");
271     }
272     if (lastCommittedTransition == default)
273     {
274         FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
275     }
276     _lastCommittedTransition = lastCommittedTransition;
277     // TODO: Think about a better way to calculate or store this value
278     var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
279     _lastCommittedTransactionId = allTransitions.Length > 0 ? allTransitions.Max(x =>
280         ↳ x.TransactionId) : 0;
281     _uniqueTimestampFactory = new UniqueTimestampFactory();
282     _logAddress = logAddress;
283     _log = FileHelpers.Append(logAddress);
284     _transitions = new Queue<Transition>();
285     _transitionsPusher = new Task(TransitionsPusher);
286     _transitionsPusher.Start();
287 }
288
289 [MethodImpl(MethodImplOptions.AggressiveInlining)]
290 public IList<ulong> GetLinkValue(ulong link) => _links.GetLink(link);
291
292 [MethodImpl(MethodImplOptions.AggressiveInlining)]
293 public override ulong Create(IList<ulong> restrictions)
294 {
295     var createdLinkIndex = _links.Create();
296     var createdLink = new Link<ulong>(_links.GetLink(createdLinkIndex));
297     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
298         ↳ default, createdLink));
299     return createdLinkIndex;
300 }
301
302 [MethodImpl(MethodImplOptions.AggressiveInlining)]
303 public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
304 {
305     var linkIndex = restrictions[_constants.IndexPart];
306     var beforeLink = new Link<ulong>(_links.GetLink(linkIndex));
307     linkIndex = _links.Update(restrictions, substitution);
308     var afterLink = new Link<ulong>(_links.GetLink(linkIndex));
309     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
310         ↳ beforeLink, afterLink));
311     return linkIndex;
312 }
313
314 [MethodImpl(MethodImplOptions.AggressiveInlining)]
315 public override void Delete(IList<ulong> restrictions)
316 {
317     var link = restrictions[_constants.IndexPart];
318     var deletedLink = new Link<ulong>(_links.GetLink(link));
319     _links.Delete(link);
320     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
321         ↳ deletedLink, default));
322 }
323
324 [MethodImpl(MethodImplOptions.AggressiveInlining)]
325 private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
326     ↳ _transitions;
327
328 [MethodImpl(MethodImplOptions.AggressiveInlining)]
329 private void CommitTransition(Transition transition)
330 {

```

```

325         if (_currentTransaction != null)
326         {
327             Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
328         }
329         var transitions = GetCurrentTransitions();
330         transitions.Enqueue(transition);
331     }
332
333     [MethodImpl(MethodImplOptions.AggressiveInlining)]
334     private void RevertTransition(Transition transition)
335     {
336         if (transition.After.IsNull()) // Revert Deletion with Creation
337         {
338             _links.Create();
339         }
340         else if (transition.Before.IsNull()) // Revert Creation with Deletion
341         {
342             _links.Delete(transition.After.Index);
343         }
344         else // Revert Update
345         {
346             _links.Update(new[] { transition.After.Index, transition.Before.Source,
347                                     ↪ transition.Before.Target });
348         }
349     }
350
351     [MethodImpl(MethodImplOptions.AggressiveInlining)]
352     private void ResetCurrentTransation()
353     {
354         _currentTransactionId = 0;
355         _currentTransactionTransitions = null;
356         _currentTransaction = null;
357     }
358
359     [MethodImpl(MethodImplOptions.AggressiveInlining)]
360     private void PushTransitions()
361     {
362         if (_log == null || _transitions == null)
363         {
364             return;
365         }
366         for (var i = 0; i < _transitions.Count; i++)
367         {
368             var transition = _transitions.Dequeue();
369
370             _log.Write(transition);
371             _lastCommittedTransition = transition;
372         }
373     }
374
375     [MethodImpl(MethodImplOptions.AggressiveInlining)]
376     private void TransitionsPusher()
377     {
378         while (!Disposable.IsDisposed && _transitionsPusher != null)
379         {
380             Thread.Sleep(DefaultPushDelay);
381             PushTransitions();
382         }
383     }
384
385     [MethodImpl(MethodImplOptions.AggressiveInlining)]
386     public Transaction BeginTransaction() => new Transaction(this);
387
388     [MethodImpl(MethodImplOptions.AggressiveInlining)]
389     private void DisposeTransitions()
390     {
391         try
392         {
393             var pusher = _transitionsPusher;
394             if (pusher != null)
395             {
396                 _transitionsPusher = null;
397                 pusher.Wait();
398             }
399             if (_transitions != null)
400             {
401                 PushTransitions();
402             }
403             _log.DisposeIfPossible();

```

```

403         FileHelpers.WriteFirst(_logAddress, _lastCommittedTransition);
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 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Converters;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Unicode
7  {
8      public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
9          ↪ IConverter<char, TLink>
10     {
11         private static readonly UncheckedConverter<char, TLink> _charToAddressConverter =
12             ↪ UncheckedConverter<char, TLink>.Default;
13
14         private readonly IConverter<TLink> _addressToNumberConverter;
15         private readonly TLink _unicodeSymbolMarker;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
19             ↪ addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
20         {
21             _addressToNumberConverter = addressToNumberConverter;
22             _unicodeSymbolMarker = unicodeSymbolMarker;
23         }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public TLink Convert(char source)
27         {
28             var unaryNumber =
29                 ↪ _addressToNumberConverter.Convert(_charToAddressConverter.Convert(source));
30             return _links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
31         }
32     }
33 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Converters;
4  using Platform.Data.Doublets.Sequences.Indexes;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Unicode
9  {
10     public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
11         ↪ IConverter<string, TLink>
12     {
13         private readonly IConverter<string, IList<TLink>> _stringToUnicodeSymbolListConverter;
14         private readonly ISequenceIndex<TLink> _index;
15         private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
16         private readonly TLink _unicodeSequenceMarker;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

18     public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
    ↪     IList<TLink>> stringToUnicodeSymbolListConverter, ISequenceIndex<TLink> index,
    ↪     IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
    ↪     unicodeSequenceMarker) : base(links)
19     {
20         _stringToUnicodeSymbolListConverter = stringToUnicodeSymbolListConverter;
21         _index = index;
22         _listToSequenceLinkConverter = listToSequenceLinkConverter;
23         _unicodeSequenceMarker = unicodeSequenceMarker;
24     }
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         _index.Add(elements);
43         var sequence = _listToSequenceLinkConverter.Convert(elements);
44         return _links.GetOrCreate(sequence, _unicodeSequenceMarker);
45     }
46 }
47 }

```

1.109 ./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  }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Globalization;

```

```

4 using System.Runtime.CompilerServices;
5 using System.Text;
6 using Platform.Data.Sequences;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Unicode
11 {
12     public class UnicodeMap
13     {
14         public static readonly ulong FirstCharLink = 1;
15         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
16         public static readonly ulong MapSize = 1 + char.MaxValue;
17
18         private readonly ILinks<ulong> _links;
19         private bool _initialized;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public UnicodeMap(ILinks<ulong> links) => _links = links;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public static UnicodeMap InitNew(ILinks<ulong> links)
26         {
27             var map = new UnicodeMap(links);
28             map.Init();
29             return map;
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public void Init()
34         {
35             if (_initialized)
36             {
37                 return;
38             }
39             _initialized = true;
40             var firstLink = _links.CreatePoint();
41             if (firstLink != FirstCharLink)
42             {
43                 _links.Delete(firstLink);
44             }
45             else
46             {
47                 for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
48                 {
49                     // From NIL to It (NIL -> Character) transformation meaning, (or infinite
50                     // ↳ amount of NIL characters before actual Character)
51                     var createdLink = _links.CreatePoint();
52                     _links.Update(createdLink, firstLink, createdLink);
53                     if (createdLink != i)
54                     {
55                         throw new InvalidOperationException("Unable to initialize UTF 16
56                         ↳ table.");
57                     }
58                 }
59             }
60         }
61
62         // 0 - null link
63         // 1 - nil character (0 character)
64         // ...
65         // 65536 (0(1) + 65535 = 65536 possible values)
66
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         public static ulong FromCharToLink(char character) => (ulong)character + 1;
69
70         [MethodImpl(MethodImplOptions.AggressiveInlining)]
71         public static char FromLinkToChar(ulong link) => (char)(link - 1);
72
73         [MethodImpl(MethodImplOptions.AggressiveInlining)]
74         public static bool IsCharLink(ulong link) => link <= MapSize;
75
76         [MethodImpl(MethodImplOptions.AggressiveInlining)]
77         public static string FromLinksToString(IList<ulong> linksList)
78         {
79             var sb = new StringBuilder();
80             for (int i = 0; i < linksList.Count; i++)
81             {
82                 sb.Append(FromLinkToChar(linksList[i]));
83             }
84         }
85     }
86 }

```



```

81     }
82     return sb.ToString();
83 }
84
85 [MethodImpl(MethodImplOptions.AggressiveInlining)]
86 public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
87 {
88     var sb = new StringBuilder();
89     if (links.Exists(link))
90     {
91         StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
92             x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
93             ↪ element =>
94             {
95                 sb.Append(FromLinkToChar(element));
96                 return true;
97             }
98     );
99     return sb.ToString();
100 }
101
102 [MethodImpl(MethodImplOptions.AggressiveInlining)]
103 public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
104     ↪ chars.Length);
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
108 {
109     // char array to ulong array
110     var linksSequence = new ulong[count];
111     for (var i = 0; i < count; i++)
112     {
113         linksSequence[i] = FromCharToLink(chars[i]);
114     }
115     return linksSequence;
116 }
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 public static ulong[] FromStringToLinkArray(string sequence)
120 {
121     // char array to ulong array
122     var linksSequence = new ulong[sequence.Length];
123     for (var i = 0; i < sequence.Length; i++)
124     {
125         linksSequence[i] = FromCharToLink(sequence[i]);
126     }
127     return linksSequence;
128 }
129
130 [MethodImpl(MethodImplOptions.AggressiveInlining)]
131 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
132 {
133     var result = new List<ulong[]>();
134     var offset = 0;
135     while (offset < sequence.Length)
136     {
137         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
138         var relativeLength = 1;
139         var absoluteLength = offset + relativeLength;
140         while (absoluteLength < sequence.Length &&
141             ↪ currentCategory == CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
142         {
143             relativeLength++;
144             absoluteLength++;
145         }
146         // char array to ulong array
147         var innerSequence = new ulong[relativeLength];
148         var maxLength = offset + relativeLength;
149         for (var i = offset; i < maxLength; i++)
150         {
151             innerSequence[i - offset] = FromCharToLink(sequence[i]);
152         }
153         result.Add(innerSequence);
154         offset += relativeLength;
155     }
156     return result;
157 }

```

```

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 =
168                 ↪ CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
169             var absoluteLength = offset + relativeLength;
170             while (absoluteLength < array.Length &&
171                 array[absoluteLength] <= LastCharLink &&
172                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
173                     ↪ array[absoluteLength])))
174             {
175                 relativeLength++;
176                 absoluteLength++;
177             }
178         }
179         else
180         {
181             var absoluteLength = offset + relativeLength;
182             while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
183             {
184                 relativeLength++;
185                 absoluteLength++;
186             }
187             // copy array
188             var innerSequence = new ulong[relativeLength];
189             var maxLength = offset + relativeLength;
190             for (var i = offset; i < maxLength; i++)
191             {
192                 innerSequence[i - offset] = array[i];
193             }
194             result.Add(innerSequence);
195             offset += relativeLength;
196         }
197     }
198     return result;
199 }

```

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

```

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>,
13         ↪ IConverter<TLink, string>
14     {
15         private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
16         private readonly ISequenceWalker<TLink> _sequenceWalker;
17         private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
21             ↪ unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
22             ↪ IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
23         {
24             _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
25             _sequenceWalker = sequenceWalker;
26             _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public string Convert(TLink source)
31         {
32             if (!_unicodeSequenceCriterionMatcher.IsMatched(source))
33             {

```

```

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 }

```

1.112 ./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 }

```

1.113 ./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.TestRowNumbersCRUDOperations());
27         Using<ulong>(links => links.TestRowNumbersCRUDOperations());
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

```

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

```

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

```

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

```

1  using Xunit;
2
3  namespace Platform.Data.Doublets.Tests
4  {
5      public static class LinksConstantsTests
6      {
7          [Fact]
8          public static void ExternalReferencesTest()
9          {
10             LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
            ↳ (long.MaxValue + 1UL, ulong.MaxValue));
11
12             //var minimum = new Hybrid<ulong>(0, isExternal: true);
13             var minimum = new Hybrid<ulong>(1, isExternal: true);
14             var maximum = new Hybrid<ulong>(long.MaxValue, isExternal: true);
15
16             Assert.True(constants.IsExternalReference(minimum));
17             Assert.True(constants.IsExternalReference(maximum));
18         }
19     }
20 }

```

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

```

1  using System;
2  using System.Linq;
3  using Xunit;
4  using Platform.Collections.Stacks;

```

```

5 using Platform.Collections.Arrays;
6 using Platform.Memory;
7 using Platform.Data.Numbers.Raw;
8 using Platform.Data.Doublets.Sequences;
9 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
10 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
11 using Platform.Data.Doublets.Sequences.Converters;
12 using Platform.Data.Doublets.PropertyOperators;
13 using Platform.Data.Doublets.Incrementers;
14 using Platform.Data.Doublets.Sequences.Walkers;
15 using Platform.Data.Doublets.Sequences.Indexes;
16 using Platform.Data.Doublets.Unicode;
17 using Platform.Data.Doublets.Numbers.Unary;
18 using Platform.Data.Doublets.Decorators;
19 using Platform.Data.Doublets.Memory.United.Specific;
20
21 namespace Platform.Data.Doublets.Tests
22 {
23     public static class OptimalVariantSequenceTests
24     {
25         private static readonly string _sequenceExample = "зеленела зелёная зелень";
26         private static readonly string _loremIpsumExample = @"Lorem ipsum dolor sit amet,
                ↳ consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore
                ↳ magna aliqua.
Facilisi nullam vehicula ipsum a arcu cursus vitae congue mauris.
Et malesuada fames ac turpis egestas sed.
Eget velit aliquet sagittis id consectetur purus.
Dignissim cras tincidunt lobortis feugiat vivamus.
Vitae aliquet nec ullamcorper sit.
Lectus quam id leo in vitae.
Tortor dignissim convallis aenean et tortor at risus viverra adipiscing.
Sed risus ultricies tristique nulla aliquet enim tortor at auctor.
Integer eget aliquet nibh praesent tristique.
Vitae congue eu consequat ac felis donec et odio.
Tristique et egestas quis ipsum suspendisse.
Suspendisse potenti nullam ac tortor vitae purus faucibus ornare.
Nulla facilisi etiam dignissim diam quis enim lobortis scelerisque.
Imperdiet proin fermentum leo vel orci.
In ante metus dictum at tempor commodo.
Nisi lacus sed viverra tellus in.
Quam vulputate dignissim suspendisse in.
Elit scelerisque mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus.
Gravida cum sociis natoque penatibus et magnis dis parturient.
Risus quis varius quam quisque id diam.
Congue nisi vitae suscipit tellus mauris a diam maecenas.
Eget nunc scelerisque viverra mauris in aliquam sem fringilla.
Pharetra vel turpis nunc eget lorem dolor sed viverra.
Mattis pellentesque id nibh tortor id aliquet.
Purus non enim praesent elementum facilisis leo vel.
Etiam sit amet nisl purus in mollis nunc sed.
Tortor at auctor urna nunc id cursus metus aliquam.
Volutpat odio facilisis mauris sit amet.
Turpis egestas pretium aenean pharetra magna ac placerat.
Fermentum dui faucibus in ornare quam viverra orci sagittis eu.
Porttitor leo a diam sollicitudin tempor id eu.
Volutpat sed cras ornare arcu dui.
Ut aliquam purus sit amet luctus venenatis lectus magna.
Aliquet risus feugiat in ante metus dictum at.
Mattis nunc sed blandit libero.
Elit pellentesque habitant morbi tristique senectus et netus.
Nibh sit amet commodo nulla facilisi nullam vehicula ipsum a.
Enim sit amet venenatis urna cursus eget nunc scelerisque viverra.
Amet venenatis urna cursus eget nunc scelerisque viverra mauris in.
Diam donec adipiscing tristique risus nec feugiat.
Pulvinar mattis nunc sed blandit libero volutpat.
Cras fermentum odio eu feugiat pretium nibh ipsum.
In nulla posuere sollicitudin aliquam ultrices sagittis orci a.
Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.
A iaculis at erat pellentesque.
Morbi blandit cursus risus at ultrices mi tempus imperdiet nulla.
Eget lorem dolor sed viverra ipsum nunc.
Leo a diam sollicitudin tempor id eu.
Interdum consectetur libero id faucibus nisl tincidunt eget nullam non.";
                ];
76
77         [Fact]
78         public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
79         {
80             using (var scope = new TempLinksTestScope(useSequences: false))
81             {
82                 var links = scope.Links;
83                 var constants = links.Constants;
84
85                 links.UseUnicode();
86

```

```

87     var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
88
89     var meaningRoot = links.CreatePoint();
90     var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
91     var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
92     var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
93         ↳ constants.Itself);
94
95     var unaryNumberToAddressConverter = new
96         ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
97     var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
98     var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
99         ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
100    var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
101        ↳ frequencyPropertyMarker, frequencyMarker);
102    var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
103        ↳ frequencyPropertyOperator, frequencyIncrementer);
104    var linkToItsFrequencyNumberConverter = new
105        ↳ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
106        ↳ unaryNumberToAddressConverter);
107    var sequenceToItsLocalElementLevelsConverter = new
108        ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
109        ↳ linkToItsFrequencyNumberConverter);
110    var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
111        ↳ sequenceToItsLocalElementLevelsConverter);
112
113    var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
114        ↳ Walker = new LeveledSequenceWalker<ulong>(links) });
115
116    ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
117        ↳ index, optimalVariantConverter);
118    }
119 }
120
121 [Fact]
122 public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
123 {
124     using (var scope = new TempLinksTestScope(useSequences: false))
125     {
126         var links = scope.Links;
127
128         links.UseUnicode();
129
130         var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
131
132         var totalSequenceSymbolFrequencyCounter = new
133             ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links);
134
135         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
136             ↳ totalSequenceSymbolFrequencyCounter);
137
138         var index = new
139             ↳ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
140         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
141
142         var sequenceToItsLocalElementLevelsConverter = new
143             ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
144             ↳ linkToItsFrequencyNumberConverter);
145         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
146             ↳ sequenceToItsLocalElementLevelsConverter);
147
148         var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
149             ↳ Walker = new LeveledSequenceWalker<ulong>(links) });
150
151         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
152             ↳ index, optimalVariantConverter);
153     }
154 }
155
156 private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
157     ↳ SequenceToItsLocalElementLevelsConverter<ulong>
158     ↳ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
159     ↳ OptimalVariantConverter<ulong> optimalVariantConverter)
160 {
161     index.Add(sequence);
162
163     var optimalVariant = optimalVariantConverter.Convert(sequence);

```

```

141     var readSequence1 = sequences.ToList(optimalVariant);
142
143     Assert.True(sequence.SequenceEqual(readSequence1));
144 }
145
146 [Fact]
147 public static void SavedSequencesOptimizationTest()
148 {
149     LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
150         ↳ (long.MaxValue + 1UL, ulong.MaxValue));
151
152     using (var memory = new HeapResizableDirectMemory())
153     using (var disposableLinks = new UInt64UnitedMemoryLinks(memory,
154         ↳ UInt64UnitedMemoryLinks.DefaultLinksSizeStep, constants, useAvlBasedIndex:
155         ↳ false))
156     {
157         var links = new UInt64Links(disposableLinks);
158
159         var root = links.CreatePoint();
160
161         //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
162         var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
163
164         var unicodeSymbolMarker = links.GetOrCreate(root,
165             ↳ addressToNumberConverter.Convert(1));
166         var unicodeSequenceMarker = links.GetOrCreate(root,
167             ↳ addressToNumberConverter.Convert(2));
168
169         var totalSequenceSymbolFrequencyCounter = new
170             ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links);
171         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
172             ↳ totalSequenceSymbolFrequencyCounter);
173         var index = new
174             ↳ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
175         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
176             ↳ ncyNumberConverter<ulong>(linkFrequenciesCache);
177         var sequenceToItsLocalElementLevelsConverter = new
178             ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
179             ↳ linkToItsFrequencyNumberConverter);
180         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
181             ↳ sequenceToItsLocalElementLevelsConverter);
182
183         var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
184             ↳ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
185
186         var unicodeSequencesOptions = new SequencesOptions<ulong>()
187         {
188             UseSequenceMarker = true,
189             SequenceMarkerLink = unicodeSequenceMarker,
190             UseIndex = true,
191             Index = index,
192             LinksToSequenceConverter = optimalVariantConverter,
193             Walker = walker,
194             UseGarbageCollection = true
195         };
196
197         var unicodeSequences = new Sequences.Sequences(new
198             ↳ SynchronizedLinks<ulong>(links), unicodeSequencesOptions);
199
200         // Create some sequences
201         var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
202             ↳ StringSplitOptions.RemoveEmptyEntries);
203         var arrays = strings.Select(x => x.Select(y =>
204             ↳ addressToNumberConverter.Convert(y)).ToArray()).ToArray();
205         for (int i = 0; i < arrays.Length; i++)
206         {
207             unicodeSequences.Create(arrays[i].ShiftRight());
208         }
209
210         var linksCountAfterCreation = links.Count();
211
212         // get list of sequences links
213         // for each sequence link
214         //     create new sequence version
215         //     if new sequence is not the same as sequence link
216         //         delete sequence link
217         //         collect garbage

```

```

203         unicodeSequences.CompactAll();
204
205         var linksCountAfterCompactification = links.Count();
206
207         Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
208     }
209 }
210 }
211 }

```

1.117 ./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

```

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

```

1  using System.IO;
2  using Xunit;

```



```

3 using Platform.Singletons;
4 using Platform.Memory;
5 using Platform.Data.Doublets.Memory.United.Specific;
6
7 namespace Platform.Data.Doublets.Tests
8 {
9     public static class ResizableDirectMemoryLinksTests
10    {
11        private static readonly LinksConstants<ulong> _constants =
12            ↳ Default<LinksConstants<ulong>>.Instance;
13
14        [Fact]
15        public static void BasicFileMappedMemoryTest()
16        {
17            var tempFilename = Path.GetTempFileName();
18            using (var memoryAdapter = new UInt64UnitedMemoryLinks(tempFilename))
19            {
20                memoryAdapter.TestBasicMemoryOperations();
21            }
22            File.Delete(tempFilename);
23
24            [Fact]
25            public static void BasicHeapMemoryTest()
26            {
27                using (var memory = new
28                    ↳ HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
29                using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
30                    ↳ UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
31                {
32                    memoryAdapter.TestBasicMemoryOperations();
33                }
34
35                private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
36                {
37                    var link = memoryAdapter.Create();
38                    memoryAdapter.Delete(link);
39
40                    [Fact]
41                    public static void NonexistentReferencesHeapMemoryTest()
42                    {
43                        using (var memory = new
44                            ↳ HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
45                        using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
46                            ↳ UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
47                        {
48                            memoryAdapter.TestNonexistentReferences();
49                        }
50
51                        private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
52                        {
53                            var link = memoryAdapter.Create();
54                            memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
55                            var resultLink = _constants.Null;
56                            memoryAdapter.Each(foundLink =>
57                                {
58                                    resultLink = foundLink[_constants.IndexPart];
59                                    return _constants.Break;
60                                }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
61                            Assert.True(resultLink == link);
62                            Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
63                            memoryAdapter.Delete(link);
64                        }
65                    }
66                }
67            }
68        }
69    }
70 }

```

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

```

1 using Xunit;
2 using Platform.Scopes;
3 using Platform.Memory;
4 using Platform.Data.Doublets.Decorators;
5 using Platform.Reflection;
6 using Platform.Data.Doublets.Memory.United.Generic;
7 using Platform.Data.Doublets.Memory.United.Specific;
8
9 namespace Platform.Data.Doublets.Tests

```

```

10 {
11     public static class ScopeTests
12     {
13         [Fact]
14         public static void SingleDependencyTest()
15         {
16             using (var scope = new Scope())
17             {
18                 scope.IncludeAssemblyOf<IMemory>();
19                 var instance = scope.Use<IDirectMemory>();
20                 Assert.IsType<HeapResizableDirectMemory>(instance);
21             }
22         }
23
24         [Fact]
25         public static void CascadeDependencyTest()
26         {
27             using (var scope = new Scope())
28             {
29                 scope.Include<TemporaryFileMappedResizableDirectMemory>();
30                 scope.Include<UInt64UnitedMemoryLinks>();
31                 var instance = scope.Use<ILinks<ulong>>();
32                 Assert.IsType<UInt64UnitedMemoryLinks>(instance);
33             }
34         }
35
36         [Fact]
37         public static void FullAutoResolutionTest()
38         {
39             using (var scope = new Scope(autoInclude: true, autoExplore: true))
40             {
41                 var instance = scope.Use<UInt64Links>();
42                 Assert.IsType<UInt64Links>(instance);
43             }
44         }
45
46         [Fact]
47         public static void TypeParametersTest()
48         {
49             using (var scope = new Scope<Types<HeapResizableDirectMemory,
50                 ↪ UnitedMemoryLinks<ulong>>>())
51             {
52                 var links = scope.Use<ILinks<ulong>>();
53                 Assert.IsType<UnitedMemoryLinks<ulong>>(links);
54             }
55         }
56     }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Linq;
5  using Xunit;
6  using Platform.Collections;
7  using Platform.Collections.Arrays;
8  using Platform.Random;
9  using Platform.IO;
10 using Platform.Singletons;
11 using Platform.Data.Doublets.Sequences;
12 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
13 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
14 using Platform.Data.Doublets.Sequences.Converters;
15 using Platform.Data.Doublets.Unicode;
16
17 namespace Platform.Data.Doublets.Tests
18 {
19     public static class SequencesTests
20     {
21         private static readonly LinksConstants<ulong> _constants =
22             ↪ Default<LinksConstants<ulong>>.Instance;
23
24         static SequencesTests()
25         {
26             // Trigger static constructor to not mess with perfomance measurements
27             _ = BitString.GetBitMaskFromIndex(1);
28         }
29
30         [Fact]

```

```

30 public static void CreateAllVariantsTest()
31 {
32     const long sequenceLength = 8;
33
34     using (var scope = new TempLinksTestScope(useSequences: true))
35     {
36         var links = scope.Links;
37         var sequences = scope.Sequences;
38
39         var sequence = new ulong[sequenceLength];
40         for (var i = 0; i < sequenceLength; i++)
41         {
42             sequence[i] = links.Create();
43         }
44
45         var sw1 = Stopwatch.StartNew();
46         var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
47
48         var sw2 = Stopwatch.StartNew();
49         var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
50
51         Assert.True(results1.Count > results2.Length);
52         Assert.True(sw1.Elapsed > sw2.Elapsed);
53
54         for (var i = 0; i < sequenceLength; i++)
55         {
56             links.Delete(sequence[i]);
57         }
58
59         Assert.True(links.Count() == 0);
60     }
61 }
62
63 //[Fact]
64 //public void CUDTest()
65 //{
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,
73 //        ↳ DefaultLinksSizeStep))
74 //    using (var links = new Links(memoryAdapter))
75 //    {
76 //        var sequence = new ulong[sequenceLength];
77 //        for (var i = 0; i < sequenceLength; i++)
78 //            sequence[i] = links.Create(itself, itself);
79 //
80 //        SequencesOptions o = new SequencesOptions();
81 //
82 //        TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
83 //        o.
84 //
85 //        var sequences = new Sequences(links);
86 //
87 //        var sw1 = Stopwatch.StartNew();
88 //        var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
89 //
90 //        var sw2 = Stopwatch.StartNew();
91 //        var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
92 //
93 //        Assert.True(results1.Count > results2.Length);
94 //        Assert.True(sw1.Elapsed > sw2.Elapsed);
95 //
96 //        for (var i = 0; i < sequenceLength; i++)
97 //            links.Delete(sequence[i]);
98 //    }
99 //
100 //    File.Delete(tempFilename);
101 //}
102
103 [Fact]
104 public static void AllVariantsSearchTest()
105 {
106     const long sequenceLength = 8;
107
108     using (var scope = new TempLinksTestScope(useSequences: true))
109     {

```

```

109     var links = scope.Links;
110     var sequences = scope.Sequences;
111
112     var sequence = new ulong[sequenceLength];
113     for (var i = 0; i < sequenceLength; i++)
114     {
115         sequence[i] = links.Create();
116     }
117
118     var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
119
120     //for (int i = 0; i < createResults.Length; i++)
121     //    sequences.Create(createResults[i]);
122
123     var sw0 = Stopwatch.StartNew();
124     var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
125
126     var sw1 = Stopwatch.StartNew();
127     var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
128
129     var sw2 = Stopwatch.StartNew();
130     var searchResults2 = sequences.Each1(sequence); sw2.Stop();
131
132     var sw3 = Stopwatch.StartNew();
133     var searchResults3 = sequences.Each(sequence.ShiftRight()); sw3.Stop();
134
135     var intersection0 = createResults.Intersect(searchResults0).ToList();
136     Assert.True(intersection0.Count == searchResults0.Count);
137     Assert.True(intersection0.Count == createResults.Length);
138
139     var intersection1 = createResults.Intersect(searchResults1).ToList();
140     Assert.True(intersection1.Count == searchResults1.Count);
141     Assert.True(intersection1.Count == createResults.Length);
142
143     var intersection2 = createResults.Intersect(searchResults2).ToList();
144     Assert.True(intersection2.Count == searchResults2.Count);
145     Assert.True(intersection2.Count == createResults.Length);
146
147     var intersection3 = createResults.Intersect(searchResults3).ToList();
148     Assert.True(intersection3.Count == searchResults3.Count);
149     Assert.True(intersection3.Count == createResults.Length);
150
151     for (var i = 0; i < sequenceLength; i++)
152     {
153         links.Delete(sequence[i]);
154     }
155 }
156
157 [Fact]
158 public static void BalancedVariantSearchTest()
159 {
160     const long sequenceLength = 200;
161
162     using (var scope = new TempLinksTestScope(useSequences: true))
163     {
164         var links = scope.Links;
165         var sequences = scope.Sequences;
166
167         var sequence = new ulong[sequenceLength];
168         for (var i = 0; i < sequenceLength; i++)
169         {
170             sequence[i] = links.Create();
171         }
172
173         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
174
175         var sw1 = Stopwatch.StartNew();
176         var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
177
178         var sw2 = Stopwatch.StartNew();
179         var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
180
181         var sw3 = Stopwatch.StartNew();
182         var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
183
184         // На количестве в 200 элементов это будет занимать вечность
185         //var sw4 = Stopwatch.StartNew();
186         //var searchResults4 = sequences.Each(sequence); sw4.Stop();
187
188     }

```

```

189     Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
190
191     Assert.True(searchResults3.Count == 1 && balancedVariant ==
192         ↳ searchResults3.First());
193
194     //Assert.True(sw1.Elapsed < sw2.Elapsed);
195
196     for (var i = 0; i < sequenceLength; i++)
197     {
198         links.Delete(sequence[i]);
199     }
200 }
201
202 [Fact]
203 public static void AllPartialVariantsSearchTest()
204 {
205     const long sequenceLength = 8;
206
207     using (var scope = new TempLinksTestScope(useSequences: true))
208     {
209         var links = scope.Links;
210         var sequences = scope.Sequences;
211
212         var sequence = new ulong[sequenceLength];
213         for (var i = 0; i < sequenceLength; i++)
214         {
215             sequence[i] = links.Create();
216         }
217
218         var createResults = sequences.CreateAllVariants2(sequence);
219
220         //var createResultsStrings = createResults.Select(x => x + ": " +
221         ↳ sequences.FormatSequence(x)).ToList();
222         //Global.Trash = createResultsStrings;
223
224         var partialSequence = new ulong[sequenceLength - 2];
225
226         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
227
228         var sw1 = Stopwatch.StartNew();
229         var searchResults1 =
230             ↳ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
231
232         var sw2 = Stopwatch.StartNew();
233         var searchResults2 =
234             ↳ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
235
236         //var sw3 = Stopwatch.StartNew();
237         //var searchResults3 =
238             ↳ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
239
240         var sw4 = Stopwatch.StartNew();
241         var searchResults4 =
242             ↳ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
243
244         //Global.Trash = searchResults3;
245
246         //var searchResults1Strings = searchResults1.Select(x => x + ": " +
247         ↳ sequences.FormatSequence(x)).ToList();
248         //Global.Trash = searchResults1Strings;
249
250         var intersection1 = createResults.Intersect(searchResults1).ToList();
251         Assert.True(intersection1.Count == createResults.Length);
252
253         var intersection2 = createResults.Intersect(searchResults2).ToList();
254         Assert.True(intersection2.Count == createResults.Length);
255
256         var intersection4 = createResults.Intersect(searchResults4).ToList();
257         Assert.True(intersection4.Count == createResults.Length);
258
259         for (var i = 0; i < sequenceLength; i++)
260         {
261             links.Delete(sequence[i]);
262         }
263     }
264 }
265
266 [Fact]

```

```

261 public static void BalancedPartialVariantsSearchTest()
262 {
263     const long sequenceLength = 200;
264
265     using (var scope = new TempLinksTestScope(useSequences: true))
266     {
267         var links = scope.Links;
268         var sequences = scope.Sequences;
269
270         var sequence = new ulong[sequenceLength];
271         for (var i = 0; i < sequenceLength; i++)
272         {
273             sequence[i] = links.Create();
274         }
275
276         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
277         var balancedVariant = balancedVariantConverter.Convert(sequence);
278
279         var partialSequence = new ulong[sequenceLength - 2];
280
281         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
282
283         var sw1 = Stopwatch.StartNew();
284         var searchResults1 =
285             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
286
287         var sw2 = Stopwatch.StartNew();
288         var searchResults2 =
289             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
290
291         Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
292         Assert.True(searchResults2.Count == 1 && balancedVariant ==
293             ↪ searchResults2.First());
294
295         for (var i = 0; i < sequenceLength; i++)
296         {
297             links.Delete(sequence[i]);
298         }
299     }
300
301     [Fact(Skip = "Correct implementation is pending")]
302     public static void PatternMatchTest()
303     {
304         var zeroOrMany = Sequences.Sequences.ZeroOrMany;
305
306         using (var scope = new TempLinksTestScope(useSequences: true))
307         {
308             var links = scope.Links;
309             var sequences = scope.Sequences;
310
311             var e1 = links.Create();
312             var e2 = links.Create();
313
314             var sequence = new[]
315             {
316                 e1, e2, e1, e2 // mama / papa
317             };
318
319             var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
320             var balancedVariant = balancedVariantConverter.Convert(sequence);
321
322             // 1: [1]
323             // 2: [2]
324             // 3: [1,2]
325             // 4: [1,2,1,2]
326
327             var doublet = links.GetSource(balancedVariant);
328
329             var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
330
331             Assert.True(matchedSequences1.Count == 0);
332
333             var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
334
335             Assert.True(matchedSequences2.Count == 0);
336
337

```

```

338     var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
339
340     Assert.True(matchedSequences3.Count == 0);
341
342     var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
343
344     Assert.Contains(doublet, matchedSequences4);
345     Assert.Contains(balancedVariant, matchedSequences4);
346
347     for (var i = 0; i < sequence.Length; i++)
348     {
349         links.Delete(sequence[i]);
350     }
351 }
352
353 [Fact]
354 public static void IndexTest()
355 {
356     using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
357         ↪ true }, useSequences: true))
358     {
359         var links = scope.Links;
360         var sequences = scope.Sequences;
361         var index = sequences.Options.Index;
362
363         var e1 = links.Create();
364         var e2 = links.Create();
365
366         var sequence = new[]
367         {
368             e1, e2, e1, e2 // mama / papa
369         };
370
371         Assert.False(index.MightContain(sequence));
372
373         index.Add(sequence);
374
375         Assert.True(index.MightContain(sequence));
376     }
377
378     /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%
379     ↪ DO%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
380     ↪ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
381     private static readonly string _exampleText =
382         ↪ @"([english
383         ↪ version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))

```

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

[[чёрное пространство, белое
 ↪ пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png
 ↪ "чёрное пространство, белое пространство")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png)

Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая
 ↪ форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?

[[чёрное пространство, чёрная
 ↪ точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png
 ↪ "чёрное пространство, чёрная
 ↪ точка")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png)

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

[[белая точка, чёрная
 ↪ точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png "белая
 ↪ точка, чёрная
 ↪ точка")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png)

А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет
 ↪ если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой?
 ↪ Гранью? Разделителем? Единицей?

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

411 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?

412

413 [![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)

414

415 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?

416

417 [![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)

418

419 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?

420

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

422

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

424

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

426

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


```

428
429 [![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности,
↳ чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://
↳ /raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png "белая обычная и
↳ направленная связи со структурой из 8 цветных элементов последовательности, чёрная
↳ типизированная связь со структурой из 8 цветных элементов последовательности")](https://raw
↳ .githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)
430
431 ...
432
433 [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-anima
↳ tion-500.gif
↳ "анимация")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
↳ -animation-500.gif)";
434
435     private static readonly string _exampleLoremIpsumText =
436         @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
↳ incididunt ut labore et dolore magna aliqua.
437 Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
↳ consequat.";
438
439 [Fact]
440 public static void CompressionTest()
441 {
442     using (var scope = new TempLinksTestScope(useSequences: true))
443     {
444         var links = scope.Links;
445         var sequences = scope.Sequences;
446
447         var e1 = links.Create();
448         var e2 = links.Create();
449
450         var sequence = new[]
451         {
452             e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
453         };
454
455         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
456         var totalSequenceSymbolFrequencyCounter = new
↳ TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
457         var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
↳ totalSequenceSymbolFrequencyCounter);
458         var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
↳ balancedVariantConverter, doubletFrequenciesCache);
459
460         var compressedVariant = compressingConverter.Convert(sequence);
461
462         // 1: [1]          (1->1) point
463         // 2: [2]          (2->2) point
464         // 3: [1,2]        (1->2) doublet
465         // 4: [1,2,1,2]    (3->3) doublet
466
467         Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
468         Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
469         Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
470         Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
471
472         var source = _constants.SourcePart;
473         var target = _constants.TargetPart;
474
475         Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
476         Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
477         Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
478         Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
479
480         // 4 - length of sequence
481         Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
↳ == sequence[0]);
482         Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
↳ == sequence[1]);
483         Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
↳ == sequence[2]);
484         Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
↳ == sequence[3]);
485     }
486 }
487
488 [Fact]

```

```

489 public static void CompressionEfficiencyTest()
490 {
491     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
        ↳ StringSplitOptions.RemoveEmptyEntries);
492     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
493     var totalCharacters = arrays.Select(x => x.Length).Sum();
494
495     using (var scope1 = new TempLinksTestScope(useSequences: true))
496     using (var scope2 = new TempLinksTestScope(useSequences: true))
497     using (var scope3 = new TempLinksTestScope(useSequences: true))
498     {
499         scope1.Links.Unsync.UseUnicode();
500         scope2.Links.Unsync.UseUnicode();
501         scope3.Links.Unsync.UseUnicode();
502
503         var balancedVariantConverter1 = new
            ↳ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
504         var totalSequenceSymbolFrequencyCounter = new
            ↳ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
505         var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
            ↳ totalSequenceSymbolFrequencyCounter);
506         var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
            ↳ balancedVariantConverter1, linkFrequenciesCache1,
            ↳ doInitialFrequenciesIncrement: false);
507
508         //var compressor2 = scope2.Sequences;
509         var compressor3 = scope3.Sequences;
510
511         var constants = Default<LinksConstants<ulong>>.Instance;
512
513         var sequences = compressor3;
514         //var meaningRoot = links.CreatePoint();
515         //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
516         //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
517         //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
            ↳ constants.Itself);
518
519         //var unaryNumberToAddressConverter = new
            ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
520         //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
            ↳ unaryOne);
521         //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
            ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
522         //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
            ↳ frequencyPropertyMarker, frequencyMarker);
523         //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
            ↳ frequencyPropertyOperator, frequencyIncrementer);
524         //var linkToItsFrequencyNumberConverter = new
            ↳ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
            ↳ unaryNumberToAddressConverter);
525
526         var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
            ↳ totalSequenceSymbolFrequencyCounter);
527
528         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<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
556     ↪ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
557
558     var initialCount2 = scope2.Links.Unsync.Count();
559
560     var sw2 = Stopwatch.StartNew();
561
562     for (int i = START; i < END; i++)
563     {
564         compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
565     }
566
567     var elapsed2 = sw2.Elapsed;
568
569     for (int i = START; i < END; i++)
570     {
571         linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
572     }
573
574     var initialCount3 = scope3.Links.Unsync.Count();
575
576     var sw3 = Stopwatch.StartNew();
577
578     for (int i = START; i < END; i++)
579     {
580         //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
581         compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
582     }
583
584     var elapsed3 = sw3.Elapsed;
585
586     Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
587     ↪ Optimal variant: {elapsed3}");
588
589     // Assert.True(elapsed1 > elapsed2);
590
591     // Checks
592     for (int i = START; i < END; i++)
593     {
594         var sequence1 = compressed1[i];
595         var sequence2 = compressed2[i];
596         var sequence3 = compressed3[i];
597
598         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
599         ↪ scope1.Links.Unsync);
600
601         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
602         ↪ scope2.Links.Unsync);
603
604         var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
605         ↪ scope3.Links.Unsync);
606
607         var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
608         ↪ link.IsPartialPoint());
609         var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
610         ↪ link.IsPartialPoint());
611         var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
612         ↪ link.IsPartialPoint());
613
614         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
615         ↪ arrays[i].Length > 3)
616         //    Assert.False(structure1 == structure2);
617         //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
618         ↪ arrays[i].Length > 3)
619         //    Assert.False(structure3 == structure2);
620
621         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
622         Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
623     }
624

```

```

615     Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
        ↳ totalCharacters);
616     Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
        ↳ totalCharacters);
617     Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
        ↳ totalCharacters);
618
619     Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
        ↳ totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
        ↳ totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
        ↳ totalCharacters}");
620
621     Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
        ↳ scope2.Links.Unsync.Count() - initialCount2);
622     Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
        ↳ scope2.Links.Unsync.Count() - initialCount2);
623
624     var duplicateProvider1 = new
        ↳ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
625     var duplicateProvider2 = new
        ↳ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
626     var duplicateProvider3 = new
        ↳ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
627
628     var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
629     var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
630     var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
631
632     var duplicates1 = duplicateCounter1.Count();
633
634     ConsoleHelpers.Debug("-----");
635
636     var duplicates2 = duplicateCounter2.Count();
637
638     ConsoleHelpers.Debug("-----");
639
640     var duplicates3 = duplicateCounter3.Count();
641
642     Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
643
644     linkFrequenciesCache1.ValidateFrequencies();
645     linkFrequenciesCache3.ValidateFrequencies();
646 }
647
648
649 [Fact]
650 public static void CompressionStabilityTest()
651 {
652     // TODO: Fix bug (do a separate test)
653     //const ulong minNumbers = 0;
654     //const ulong maxNumbers = 1000;
655
656     const ulong minNumbers = 10000;
657     const ulong maxNumbers = 12500;
658
659     var strings = new List<string>();
660
661     for (ulong i = minNumbers; i < maxNumbers; i++)
662     {
663         strings.Add(i.ToString());
664     }
665
666     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
667     var totalCharacters = arrays.Select(x => x.Length).Sum();
668
669     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
        ↳ SequencesOptions<ulong> { UseCompression = true,
        ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
        using (var scope2 = new TempLinksTestScope(useSequences: true))
        {
670         scope1.Links.UseUnicode();
671         scope2.Links.UseUnicode();
672
673         //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
674         var compressor1 = scope1.Sequences;
675         var compressor2 = scope2.Sequences;
676
677         var compressed1 = new ulong[arrays.Length];
678         var compressed2 = new ulong[arrays.Length];

```

```

681
682 var sw1 = Stopwatch.StartNew();
683
684 var START = 0;
685 var END = arrays.Length;
686
687 // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
688 // Stability issue starts at 10001 or 11000
689 //for (int i = START; i < END; i++)
690 //{
691 //    var first = compressor1.Compress(arrays[i]);
692 //    var second = compressor1.Compress(arrays[i]);
693
694 //    if (first == second)
695 //        compressed1[i] = first;
696 //    else
697 //    {
698 //        // TODO: Find a solution for this case
699 //    }
700 //}
701
702 for (int i = START; i < END; i++)
703 {
704     var first = compressor1.Create(arrays[i].ShiftRight());
705     var second = compressor1.Create(arrays[i].ShiftRight());
706
707     if (first == second)
708     {
709         compressed1[i] = first;
710     }
711     else
712     {
713         // TODO: Find a solution for this case
714     }
715 }
716
717 var elapsed1 = sw1.Elapsed;
718
719 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
720
721 var sw2 = Stopwatch.StartNew();
722
723 for (int i = START; i < END; i++)
724 {
725     var first = balancedVariantConverter.Convert(arrays[i]);
726     var second = balancedVariantConverter.Convert(arrays[i]);
727
728     if (first == second)
729     {
730         compressed2[i] = first;
731     }
732 }
733
734 var elapsed2 = sw2.Elapsed;
735
736 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
737 ↪ {elapsed2}");
738
739 Assert.True(elapsed1 > elapsed2);
740
741 // Checks
742 for (int i = START; i < END; i++)
743 {
744     var sequence1 = compressed1[i];
745     var sequence2 = compressed2[i];
746
747     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
748     {
749         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
750 ↪ scope1.Links);
751
752         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
753 ↪ scope2.Links);
754
755         //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
756 ↪ link.IsPartialPoint());
757         //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
758 ↪ link.IsPartialPoint());

```

```

755         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
756             ↪ arrays[i].Length > 3)
757         //    Assert.False(structure1 == structure2);
758         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
759     }
760 }
761
762 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
763 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
764
765 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
766     ↪ totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
767     ↪ totalCharacters}");
768
769 Assert.True(scope1.Links.Count() <= scope2.Links.Count());
770 //compressor1.ValidateFrequencies();
771 }
772 }
773 [Fact]
774 public static void RandomNumbersCompressionQualityTest()
775 {
776     const ulong N = 500;
777
778     //const ulong minNumbers = 10000;
779     //const ulong maxNumbers = 20000;
780
781     //var strings = new List<string>();
782
783     //for (ulong i = 0; i < N; i++)
784     //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
785     ↪ maxNumbers).ToString());
786
787     var strings = new List<string>();
788
789     for (ulong i = 0; i < N; i++)
790     {
791         strings.Add(RandomHelpers.Default.NextUInt64().ToString());
792     }
793
794     strings = strings.Distinct().ToList();
795
796     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
797     var totalCharacters = arrays.Select(x => x.Length).Sum();
798
799     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
800     ↪ SequencesOptions<ulong> { UseCompression = true,
801     ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
802     using (var scope2 = new TempLinksTestScope(useSequences: true))
803     {
804         scope1.Links.UseUnicode();
805         scope2.Links.UseUnicode();
806
807         var compressor1 = scope1.Sequences;
808         var compressor2 = scope2.Sequences;
809
810         var compressed1 = new ulong[arrays.Length];
811         var compressed2 = new ulong[arrays.Length];
812
813         var sw1 = Stopwatch.StartNew();
814
815         var START = 0;
816         var END = arrays.Length;
817
818         for (int i = START; i < END; i++)
819         {
820             compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
821         }
822
823         var elapsed1 = sw1.Elapsed;
824
825         var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
826
827         var sw2 = Stopwatch.StartNew();
828
829         for (int i = START; i < END; i++)
830         {
831             compressed2[i] = balancedVariantConverter.Convert(arrays[i]);

```

```

829     }
830
831     var elapsed2 = sw2.Elapsed;
832
833     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
834         ↳ {elapsed2}");
835
836     Assert.True(elapsed1 > elapsed2);
837
838     // Checks
839     for (int i = START; i < END; i++)
840     {
841         var sequence1 = compressed1[i];
842         var sequence2 = compressed2[i];
843
844         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
845         {
846             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
847                 ↳ scope1.Links);
848
849             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
850                 ↳ scope2.Links);
851
852             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
853         }
854     }
855
856     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
857     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
858
859     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
860         ↳ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
861         ↳ totalCharacters}}");
862
863     // Can be worse than balanced variant
864     //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
865
866     //compressor1.ValidateFrequencies();
867 }
868
869 [Fact]
870 public static void AllTreeBreakDownAtSequencesCreationBugTest()
871 {
872     // Made out of AllPossibleConnectionsTest test.
873
874     //const long sequenceLength = 5; //100% bug
875     const long sequenceLength = 4; //100% bug
876     //const long sequenceLength = 3; //100% _no_bug (ok)
877
878     using (var scope = new TempLinksTestScope(useSequences: true))
879     {
880         var links = scope.Links;
881         var sequences = scope.Sequences;
882
883         var sequence = new ulong[sequenceLength];
884         for (var i = 0; i < sequenceLength; i++)
885         {
886             sequence[i] = links.Create();
887         }
888
889         var createResults = sequences.CreateAllVariants2(sequence);
890
891         Global.Trash = createResults;
892
893         for (var i = 0; i < sequenceLength; i++)
894         {
895             links.Delete(sequence[i]);
896         }
897     }
898 }
899
900 [Fact]
901 public static void AllPossibleConnectionsTest()
902 {
903     const long sequenceLength = 5;
904
905     using (var scope = new TempLinksTestScope(useSequences: true))
906     {

```

```

903     var links = scope.Links;
904     var sequences = scope.Sequences;
905
906     var sequence = new ulong[sequenceLength];
907     for (var i = 0; i < sequenceLength; i++)
908     {
909         sequence[i] = links.Create();
910     }
911
912     var createResults = sequences.CreateAllVariants2(sequence);
913     var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
914
915     for (var i = 0; i < 1; i++)
916     {
917         var sw1 = Stopwatch.StartNew();
918         var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
919
920         var sw2 = Stopwatch.StartNew();
921         var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
922
923         var sw3 = Stopwatch.StartNew();
924         var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
925
926         var sw4 = Stopwatch.StartNew();
927         var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
928
929         Global.Trash = searchResults3;
930         Global.Trash = searchResults4; //-V3008
931
932         var intersection1 = createResults.Intersect(searchResults1).ToList();
933         Assert.True(intersection1.Count == createResults.Length);
934
935         var intersection2 = reverseResults.Intersect(searchResults1).ToList();
936         Assert.True(intersection2.Count == reverseResults.Length);
937
938         var intersection0 = searchResults1.Intersect(searchResults2).ToList();
939         Assert.True(intersection0.Count == searchResults2.Count);
940
941         var intersection3 = searchResults2.Intersect(searchResults3).ToList();
942         Assert.True(intersection3.Count == searchResults3.Count);
943
944         var intersection4 = searchResults3.Intersect(searchResults4).ToList();
945         Assert.True(intersection4.Count == searchResults4.Count);
946     }
947
948     for (var i = 0; i < sequenceLength; i++)
949     {
950         links.Delete(sequence[i]);
951     }
952 }
953
954 [Fact(Skip = "Correct implementation is pending")]
955 public static void CalculateAllUsagesTest()
956 {
957     const long sequenceLength = 3;
958
959     using (var scope = new TempLinksTestScope(useSequences: true))
960     {
961         var links = scope.Links;
962         var sequences = scope.Sequences;
963
964         var sequence = new ulong[sequenceLength];
965         for (var i = 0; i < sequenceLength; i++)
966         {
967             sequence[i] = links.Create();
968         }
969
970         var createResults = sequences.CreateAllVariants2(sequence);
971
972         //var reverseResults =
973         ↪ sequences.CreateAllVariants2(sequence.Reverse().ToArray());
974
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         sequences.CalculateAllUsages2(linksTotalUsages2);
983
984         var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
985         Assert.True(intersection1.Count == linksTotalUsages2.Length);
986     }
987
988     for (var i = 0; i < sequenceLength; i++)
989     {
990         links.Delete(sequence[i]);
991     }
992 }
993 }
994 }
995 }
996 }

```

1.121 ./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
19         [Fact]
20         public static void RawNumbersCRUDTest()
21         {
22             UsingWithExternalReferences<byte>(links => links.TestRawNumbersCRUDOperations());
23             UsingWithExternalReferences<ushort>(links => links.TestRawNumbersCRUDOperations());
24             UsingWithExternalReferences<uint>(links => links.TestRawNumbersCRUDOperations());
25             UsingWithExternalReferences<ulong>(links => links.TestRawNumbersCRUDOperations());
26         }
27
28         [Fact]
29         public static void MultipleRandomCreationsAndDeletionsTest()
30         {
31             Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
32                 ↪ MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
33                 ↪ implementation of tree cuts out 5 bits from the address space.
34             Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
35                 ↪ stMultipleRandomCreationsAndDeletions(100));
36             Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
37                 ↪ MultipleRandomCreationsAndDeletions(100));
38             Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
39                 ↪ tMultipleRandomCreationsAndDeletions(100));
40         }
41
42         private static void Using<TLink>(Action<ILinks<TLink>> action)
43         {
44             using (var dataMemory = new HeapResizableDirectMemory())
45             using (var indexMemory = new HeapResizableDirectMemory())
46             using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory))
47             {
48                 action(memory);
49             }
50         }
51
52         private static void UsingWithExternalReferences<TLink>(Action<ILinks<TLink>> action)
53         {
54             var constants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
55             using (var dataMemory = new HeapResizableDirectMemory())
56             using (var indexMemory = new HeapResizableDirectMemory())
57             using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory,
58                 ↪ SplitMemoryLinks<TLink>.DefaultLinksSizeStep, constants))
59             {
60                 action(memory);
61             }
62         }
63     }
64 }

```

```

56     }
57 }
58 }

```

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

```

1  using System.IO;
2  using Platform.Disposables;
3  using Platform.Data.Doublets.Sequences;
4  using Platform.Data.Doublets.Decorators;
5  using Platform.Data.Doublets.Memory.United.Specific;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public class TempLinksTestScope : DisposableBase
10     {
11         public ILinks<ulong> MemoryAdapter { get; }
12         public SynchronizedLinks<ulong> Links { get; }
13         public Sequences.Sequences Sequences { get; }
14         public string TempFilename { get; }
15         public string TempTransactionLogFilename { get; }
16         private readonly bool _deleteFiles;
17
18         public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
19             ↪ useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
20             ↪ useLog) { }
21
22         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
23             ↪ true, bool useSequences = false, bool useLog = false)
24         {
25             _deleteFiles = deleteFiles;
26             TempFilename = Path.GetTempFileName();
27             TempTransactionLogFilename = Path.GetTempFileName();
28             var coreMemoryAdapter = new UInt64UnitedMemoryLinks(TempFilename);
29             MemoryAdapter = useLog ? (ILinks<ulong>)new
30                 ↪ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
31                 ↪ coreMemoryAdapter;
32             Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
33             if (useSequences)
34             {
35                 Sequences = new Sequences.Sequences(Links, sequencesOptions);
36             }
37         }
38
39         protected override void Dispose(bool manual, bool wasDisposed)
40         {
41             if (!wasDisposed)
42             {
43                 Links.Unsync.DisposeIfPossible();
44                 if (_deleteFiles)
45                 {
46                     DeleteFiles();
47                 }
48             }
49         }
50
51         public void DeleteFiles()
52         {
53             File.Delete(TempFilename);
54             File.Delete(TempTransactionLogFilename);
55         }
56     }
57 }

```

1.123 ./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     }
18 }

```

```

16     var equalityComparer = EqualityComparer<T>.Default;
17
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     links.Update(linkAddress1, h106E, h108E);
97
98
99     var link1 = new Link<T>(links.GetLink(linkAddress1));
100
101     Assert.True(equalityComparer.Equals(link1.Source, h106E));
102     Assert.True(equalityComparer.Equals(link1.Target, h108E));
103
104     // Create Link (Internal -> External)
105     var linkAddress2 = links.Create();
106
107     links.Update(linkAddress2, linkAddress1, h108E);
108
109     var link2 = new Link<T>(links.GetLink(linkAddress2));
110
111     Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
112     Assert.True(equalityComparer.Equals(link2.Target, h108E));
113
114     // Create Link (Internal -> Internal)
115     var linkAddress3 = links.Create();
116
117     links.Update(linkAddress3, linkAddress1, linkAddress2);
118
119     var link3 = new Link<T>(links.GetLink(linkAddress3));
120
121     Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
122     Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
123
124     // Search for created link
125     var setter1 = new Setter<T>(constants.Null);
126     links.Each(h106E, h108E, setter1.SetAndReturnFalse);
127
128     Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
129
130     // Search for nonexistent link
131     var setter2 = new Setter<T>(constants.Null);
132     links.Each(h106E, h107E, setter2.SetAndReturnFalse);
133
134     Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
135
136     // Update link to reference null (prepare for delete)
137     var updated = links.Update(linkAddress3, constants.Null, constants.Null);
138
139     Assert.True(equalityComparer.Equals(updated, linkAddress3));
140
141     link3 = new Link<T>(links.GetLink(linkAddress3));
142
143     Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
144     Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
145
146     // Delete link
147     links.Delete(linkAddress3);
148
149     Assert.True(equalityComparer.Equals(links.Count(), two));
150
151     var setter3 = new Setter<T>(constants.Null);
152     links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
153
154     Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
155 }
156
157 public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
→ links, int maximumOperationsPerCycle)
158 {
159     var comparer = Comparer<TLink>.Default;
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 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.IO;
5  using System.Text;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Xunit;
9  using Platform.Disposables;
10 using Platform.Ranges;
11 using Platform.Random;
12 using Platform.Timestamps;
13 using Platform.Reflection;
14 using Platform.Singletons;
15 using Platform.Scopes;
16 using Platform.Counters;
17 using Platform.Diagnostics;
18 using Platform.IO;
19 using Platform.Memory;
20 using Platform.Data.Doublets.Decorators;
21 using Platform.Data.Doublets.Memory.United.Specific;
22
23 namespace Platform.Data.Doublets.Tests
24 {
25     public static class UInt64LinksTests
26     {
27         private static readonly LinksConstants<ulong> _constants =
    ↪ Default<LinksConstants<ulong>>.Instance;
28
29         private const long Iterations = 10 * 1024;
30
31         #region Concept
32
33         [Fact]
34         public static void MultipleCreateAndDeleteTest()
35         {
36             using (var scope = new Scope<Types<HeapResizableDirectMemory,
    ↪ UInt64UnitedMemoryLinks>>())
37             {
38                 new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeleti_
    ↪ ons(100);
39             }
40         }
41
42         [Fact]

```

```

43 public static void CascadeUpdateTest()
44 {
45     var itself = _constants.Itself;
46     using (var scope = new TempLinksTestScope(useLog: true))
47     {
48         var links = scope.Links;
49
50         var l1 = links.Create();
51         var l2 = links.Create();
52
53         l2 = links.Update(l2, l2, l1, l2);
54
55         links.CreateAndUpdate(l2, itself);
56         links.CreateAndUpdate(l2, itself);
57
58         l2 = links.Update(l2, l1);
59
60         links.Delete(l2);
61
62         Global.Trash = links.Count();
63
64         links.Unsync.DisposeIfPossible(); // Close links to access log
65
66         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
        ↳ e.TempTransactionLogFilename);
67     }
68 }
69
70 [Fact]
71 public static void BasicTransactionLogTest()
72 {
73     using (var scope = new TempLinksTestScope(useLog: true))
74     {
75         var links = scope.Links;
76         var l1 = links.Create();
77         var l2 = links.Create();
78
79         Global.Trash = links.Update(l2, l2, l1, l2);
80
81         links.Delete(l1);
82
83         links.Unsync.DisposeIfPossible(); // Close links to access log
84
85         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
        ↳ e.TempTransactionLogFilename);
86     }
87 }
88
89 [Fact]
90 public static void TransactionAutoRevertedTest()
91 {
92     // Auto Reverted (Because no commit at transaction)
93     using (var scope = new TempLinksTestScope(useLog: true))
94     {
95         var links = scope.Links;
96         var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
97         using (var transaction = transactionsLayer.BeginTransaction())
98         {
99             var l1 = links.Create();
100             var l2 = links.Create();
101
102             links.Update(l2, l2, l1, l2);
103         }
104
105         Assert.Equal(OUL, links.Count());
106
107         links.Unsync.DisposeIfPossible();
108
109         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
        ↳ e.TempTransactionLogFilename);
110         Assert.Single(transitions);
111     }
112 }
113
114 [Fact]
115 public static void TransactionUserCodeErrorNoDataSavedTest()
116 {
117     // User Code Error (Autoreverted), no data saved
118     var itself = _constants.Itself;

```

```

119 TempLinksTestScope lastScope = null;
120 try
121 {
122     using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
123         ↪ useLog: true))
124     {
125         var links = scope.Links;
126         var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor_
127             ↪ atorBase<ulong>)links.Unsync).Links;
128         using (var transaction = transactionsLayer.BeginTransaction())
129         {
130             var l1 = links.CreateAndUpdate(itself, itself);
131             var l2 = links.CreateAndUpdate(itself, itself);
132
133             l2 = links.Update(l2, l2, l1, l2);
134
135             links.CreateAndUpdate(l2, itself);
136             links.CreateAndUpdate(l2, itself);
137
138             //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi_
139             ↪ tion>(scope.TempTransactionLogFilename);
140
141             l2 = links.Update(l2, l1);
142
143             links.Delete(l2);
144
145             ExceptionThrower();
146
147             transaction.Commit();
148         }
149         Global.Trash = links.Count();
150     }
151 catch
152 {
153     Assert.False(lastScope == null);
154
155     var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l_
156         ↪ astScope.TempTransactionLogFilename);
157
158     Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
159         ↪ transitions[0].After.IsNull());
160
161     lastScope.DeleteFiles();
162 }
163
164 [Fact]
165 public static void TransactionUserCodeErrorSomeDataSavedTest()
166 {
167     // User Code Error (Autoreverted), some data saved
168     var itself = _constants.Itself;
169
170     TempLinksTestScope lastScope = null;
171     try
172     {
173         ulong l1;
174         ulong l2;
175
176         using (var scope = new TempLinksTestScope(useLog: true))
177         {
178             var links = scope.Links;
179             l1 = links.CreateAndUpdate(itself, itself);
180             l2 = links.CreateAndUpdate(itself, itself);
181
182             l2 = links.Update(l2, l2, l1, l2);
183
184             links.CreateAndUpdate(l2, itself);
185             links.CreateAndUpdate(l2, itself);
186
187             links.Unsync.DisposeIfPossible();
188
189             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
190                 ↪ scope.TempTransactionLogFilename);
191
192         }
193
194         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
195             ↪ useLog: true))

```

```

192     {
193         var links = scope.Links;
194         var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
195         using (var transaction = transactionsLayer.BeginTransaction())
196         {
197             l2 = links.Update(l2, l1);
198
199             links.Delete(l2);
200
201             ExceptionThrower();
202
203             transaction.Commit();
204         }
205
206         Global.Trash = links.Count();
207     }
208 }
209 catch
210 {
211     Assert.False(lastScope == null);
212
213     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
        ↪ Scope.TempTransactionLogFilename);
214
215     lastScope.DeleteFiles();
216 }
217 }
218
219 [Fact]
220 public static void TransactionCommit()
221 {
222     var itself = _constants.Itself;
223
224     var tempDatabaseFilename = Path.GetTempFileName();
225     var tempTransactionLogFilename = Path.GetTempFileName();
226
227     // Commit
228     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
        ↪ 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         links.Delete(l1);
268
269         transaction.Commit();
270     }
271
272     Global.Trash = links.Count();
273 }
274
275 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
276     ↪ sactionLogFilename);
277
278 // Damage database
279
280 FileHelpers.WriteFirst(tempTransactionLogFilename, new
281     ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
282
283 // Try load damaged database
284 try
285 {
286     // TODO: Fix
287     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
288         ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
289     using (var links = new UInt64Links(memoryAdapter))
290     {
291         Global.Trash = links.Count();
292     }
293 }
294 catch (NotSupportedException ex)
295 {
296     Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
297         ↪ yet.");
298 }
299
300 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
301     ↪ sactionLogFilename);
302
303 File.Delete(tempDatabaseFilename);
304 File.Delete(tempTransactionLogFilename);
305 }
306
307 [Fact]
308 public static void Bug1Test()
309 {
310     var tempDatabaseFilename = Path.GetTempFileName();
311     var tempTransactionLogFilename = Path.GetTempFileName();
312
313     var itself = _constants.Itself;
314
315     // User Code Error (Autoreverted), some data saved
316     try
317     {
318         ulong l1;
319         ulong l2;
320
321         using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
322         using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
323             ↪ tempTransactionLogFilename))
324         using (var links = new UInt64Links(memoryAdapter))
325         {
326             l1 = links.CreateAndUpdate(itself, itself);
327             l2 = links.CreateAndUpdate(itself, itself);
328
329             l2 = links.Update(l2, l2, l1, l2);
330
331             links.CreateAndUpdate(l2, itself);
332             links.CreateAndUpdate(l2, itself);
333         }
334
335         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp_
336             ↪ TransactionLogFilename);
337
338         using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
339         using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
340             ↪ tempTransactionLogFilename))
341         using (var links = new UInt64Links(memoryAdapter))
342         {
343             using (var transaction = memoryAdapter.BeginTransaction())
344             {

```

```

338         l2 = links.Update(l2, l1);
339
340         links.Delete(l2);
341
342         ExceptionThrower();
343
344         transaction.Commit();
345     }
346
347     Global.Trash = links.Count();
348 }
349
350 catch
351 {
352     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp_
        ↪ TransactionLogFilename);
353 }
354
355 File.Delete(tempDatabaseFilename);
356 File.Delete(tempTransactionLogFilename);
357 }
358
359 private static void ExceptionThrower() => throw new InvalidOperationException();
360
361 [Fact]
362 public static void PathsTest()
363 {
364     var source = _constants.SourcePart;
365     var target = _constants.TargetPart;
366
367     using (var scope = new TempLinksTestScope())
368     {
369         var links = scope.Links;
370         var l1 = links.CreatePoint();
371         var l2 = links.CreatePoint();
372
373         var r1 = links.GetByKeys(l1, source, target, source);
374         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
375     }
376 }
377
378 [Fact]
379 public static void RecursiveStringFormattingTest()
380 {
381     using (var scope = new TempLinksTestScope(useSequences: true))
382     {
383         var links = scope.Links;
384         var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
385
386         var a = links.CreatePoint();
387         var b = links.CreatePoint();
388         var c = links.CreatePoint();
389
390         var ab = links.GetOrCreate(a, b);
391         var cb = links.GetOrCreate(c, b);
392         var ac = links.GetOrCreate(a, c);
393
394         a = links.Update(a, c, b);
395         b = links.Update(b, a, c);
396         c = links.Update(c, a, b);
397
398         Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
399         Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
400         Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
401
402         Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
            ↪ "(5:(4:5 (6:5 4)) 6)");
403         Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
            ↪ "(6:(5:(4:5 6) 6) 4)");
404         Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
            ↪ "(4:(5:4 (6:5 4)) 6)");
405
406         // TODO: Think how to build balanced syntax tree while formatting structure (eg.
            ↪ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)"
407
408         Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
            ↪ "{5}{5}{4}{6}");
409         Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
            ↪ "{5}{6}{6}{4}");

```

```

410         Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
411             ↪ "{4}{5}{4}{6}");
412     }
413 }
414 private static void DefaultFormatter(StringBuilder sb, ulong link)
415 {
416     sb.Append(link.ToString());
417 }
418 #endregion
419 #region Performance
420
421 /*
422 public static void RunAllPerformanceTests()
423 {
424     try
425     {
426         links.TestLinksInSteps();
427     }
428     catch (Exception ex)
429     {
430         ex.WriteToConsole();
431     }
432     return;
433
434     try
435     {
436         //ThreadPool.SetMaxThreads(2, 2);
437         ↪ результат // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
438         // Также это дополнительно помогает в отладке
439         // Увеличивает вероятность попадания информации в кэши
440         for (var i = 0; i < 10; i++)
441         {
442             //0 - 10 ГБ
443             //Каждые 100 МБ срез цифр
444
445             //links.TestGetSourceFunction();
446             //links.TestGetSourceFunctionInParallel();
447             //links.TestGetTargetFunction();
448             //links.TestGetTargetFunctionInParallel();
449             links.Create64BillionLinks();
450
451             links.TestRandomSearchFixed();
452             //links.Create64BillionLinksInParallel();
453             links.TestEachFunction();
454             //links.TestForeach();
455             //links.TestParallelForeach();
456         }
457
458         links.TestDeletionOfAllLinks();
459     }
460     catch (Exception ex)
461     {
462         ex.WriteToConsole();
463     }
464 }*/
465
466 /*
467 public static void TestLinksInSteps()
468 {
469     const long gibibyte = 1024 * 1024 * 1024;
470     const long mebibyte = 1024 * 1024;
471
472     var totalLinksToCreate = gibibyte /
473     ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
474     var linksStep = 102 * mebibyte /
475     ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
476
477     var creationMeasurements = new List<TimeSpan>();
478     var searchMeasurements = new List<TimeSpan>();
479     var deletionMeasurements = new List<TimeSpan>();
480
481     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         searchMeasurements.Add(Measure(() => links.RunRandomSearches(linksStep)));
497
498         Console.WriteLine("\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.WriteLine("\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],
↵ searchMeasurements[i], deletionMeasurements[i]);
517     }
518
519     ConsoleHelpers.Debug("C S D (no overhead)");
520
521     for (int i = 0; i < loops; i++)
522     {
523         ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
↵ searchMeasurements[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}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
584     }
585 }
586
587 [Fact(Skip = "performance test")]
588 public static void GetSourceInParallel()
589 {
590     using (var scope = new TempLinksTestScope())
591     {
592         var links = scope.Links;
593         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
        ↪ parallel.", Iterations);
594
595         long counter = 0;
596
597         //var firstLink = links.First();
598         var firstLink = links.Create();
599
600         var sw = Stopwatch.StartNew();
601
602         // Тестируем саму функцию
603         Parallel.For(0, Iterations, x =>
604         {
605             Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
606             //Interlocked.Increment(ref counter);
607         });
608
609         var elapsedTime = sw.Elapsed;
610
611         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
612
613         links.Delete(firstLink);
614
615         ConsoleHelpers.Debug(
616             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
        ↪ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
617     }
618 }
619
620 [Fact(Skip = "performance test")]
621 public static void TestGetTarget()
622 {
623     using (var scope = new TempLinksTestScope())
624     {
625         var links = scope.Links;
626         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
        ↪ Iterations);
627
628         ulong counter = 0;
629
630         //var firstLink = links.First();
631
632         //var firstLink = links.First();

```

```

633     var firstLink = links.Create();
634
635     var sw = Stopwatch.StartNew();
636
637     for (ulong i = 0; i < Iterations; i++)
638     {
639         counter += links.GetTarget(firstLink);
640     }
641
642     var elapsedTime = sw.Elapsed;
643
644     var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
645
646     links.Delete(firstLink);
647
648     ConsoleHelpers.Debug(
649         "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↪ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
650
651     }
652 }
653
654 [Fact(Skip = "performance test")]
655 public static void TestGetTargetInParallel()
656 {
657     using (var scope = new TempLinksTestScope())
658     {
659         var links = scope.Links;
660         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
        ↪ parallel.", Iterations);
661
662         long counter = 0;
663
664         //var firstLink = links.First();
665         var firstLink = links.Create();
666
667         var sw = Stopwatch.StartNew();
668
669         Parallel.For(0, Iterations, x =>
670         {
671             Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
672             //Interlocked.Increment(ref counter);
673         });
674
675         var elapsedTime = sw.Elapsed;
676
677         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
678
679         links.Delete(firstLink);
680
681         ConsoleHelpers.Debug(
682             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
            ↪ second), counter result: {3}",
            Iterations, elapsedTime, (long)iterationsPerSecond, counter);
683
684     }
685 }
686
687 // TODO: Заполнить базу данных перед тестом
688 /*
689 [Fact]
690 public void TestRandomSearchFixed()
691 {
692     var tempFilename = Path.GetTempFileName();
693
694     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
        ↪ DefaultLinksSizeStep))
695     {
696         long iterations = 64 * 1024 * 1024 /
        ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
697
698         ulong counter = 0;
699         var maxLink = links.Total;
700
701         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
702
703         var sw = Stopwatch.StartNew();
704
705         for (var i = iterations; i > 0; i--)
706         {

```

```

707         var source =
↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
708         var target =
↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
709
710         counter += links.Search(source, target);
711     }
712
713     var elapsedTime = sw.Elapsed;
714
715     var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
716
717     ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
↪ counter);
718 }
719
720     File.Delete(tempFilename);
721 }*/
722
723 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
724 public static void TestRandomSearchAll()
725 {
726     using (var scope = new TempLinksTestScope())
727     {
728         var links = scope.Links;
729         ulong counter = 0;
730
731         var maxLink = links.Count();
732
733         var iterations = links.Count();
734
735         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
↪ links.Count());
736
737         var sw = Stopwatch.StartNew();
738
739         for (var i = iterations; i > 0; i--)
740         {
741             var linksAddressRange = new
↪ Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);
742
743             var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
744             var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
745
746             counter += links.SearchOrDefault(source, target);
747         }
748
749         var elapsedTime = sw.Elapsed;
750
751         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
752
753         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪ Iterations per second), c: {3}",
↪ iterations, elapsedTime, (long)iterationsPerSecond, counter);
754     }
755 }
756
757 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
758 public static void TestEach()
759 {
760     using (var scope = new TempLinksTestScope())
761     {
762         var links = scope.Links;
763
764         var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
765
766         ConsoleHelpers.Debug("Testing Each function.");
767
768         var sw = Stopwatch.StartNew();
769
770         links.Each(counter.IncrementAndReturnTrue);
771
772         var elapsedTime = sw.Elapsed;
773
774         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
775
776         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
↪ links per second)",
↪ counter, elapsedTime, (long)linksPerSecond);
777
778     }

```

```

779     }
780 }
781
782 /*
783 [Fact]
784 public static void TestForeach()
785 {
786     var tempFilename = Path.GetTempFileName();
787
788     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪ DefaultLinksSizeStep))
789     {
790         ulong counter = 0;
791
792         ConsoleHelpers.Debug("Testing foreach through links.");
793
794         var sw = Stopwatch.StartNew();
795
796         //foreach (var link in links)
797         //{
798             counter++;
799         //}
800
801         var elapsedTime = sw.Elapsed;
802
803         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
804
805         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
↪ links per second)", counter, elapsedTime, (long)linksPerSecond);
806     }
807
808     File.Delete(tempFilename);
809 }
810 */
811
812 /*
813 [Fact]
814 public static void TestParallelForeach()
815 {
816     var tempFilename = Path.GetTempFileName();
817
818     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪ DefaultLinksSizeStep))
819     {
820
821         long counter = 0;
822
823         ConsoleHelpers.Debug("Testing parallel foreach through links.");
824
825         var sw = Stopwatch.StartNew();
826
827         //Parallel.ForEach((IEnumerable<ulong>)links, x =>
828         //{
829             Interlocked.Increment(ref counter);
830         //});
831
832         var elapsedTime = sw.Elapsed;
833
834         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
835
836         ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↪ {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
837     }
838
839     File.Delete(tempFilename);
840 }
841 */
842
843 [Fact(Skip = "performance test")]
844 public static void Create64BillionLinks()
845 {
846     using (var scope = new TempLinksTestScope())
847     {
848         var links = scope.Links;
849         var linksBeforeTest = links.Count();
850
851         long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;
852
853         ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
854

```



```

855     var elapsedTime = Performance.Measure(() =>
856     {
857         for (long i = 0; i < linksToCreate; i++)
858         {
859             links.Create();
860         }
861     });
862
863     var linksCreated = links.Count() - linksBeforeTest;
864     var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
865
866     ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
867
868     ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
869         ↪ linksCreated, elapsedTime,
870         (long)linksPerSecond);
871 }
872
873 [Fact(Skip = "performance test")]
874 public static void Create64BillionLinksInParallel()
875 {
876     using (var scope = new TempLinksTestScope())
877     {
878         var links = scope.Links;
879         var linksBeforeTest = links.Count();
880
881         var sw = Stopwatch.StartNew();
882
883         long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;
884
885         ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
886
887         Parallel.For(0, linksToCreate, x => links.Create());
888
889         var elapsedTime = sw.Elapsed;
890
891         var linksCreated = links.Count() - linksBeforeTest;
892         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
893
894         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
895             ↪ linksCreated, elapsedTime,
896             (long)linksPerSecond);
897     }
898
899 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
900 public static void TestDeletionOfAllLinks()
901 {
902     using (var scope = new TempLinksTestScope())
903     {
904         var links = scope.Links;
905         var linksBeforeTest = links.Count();
906
907         ConsoleHelpers.Debug("Deleting all links");
908
909         var elapsedTime = Performance.Measure(links.DeleteAll);
910
911         var linksDeleted = linksBeforeTest - links.Count();
912         var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
913
914         ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
915             ↪ linksDeleted, elapsedTime,
916             (long)linksPerSecond);
917     }
918 }
919 #endregion
920 }
921 }

```

1.125 ./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
19                 ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
20             var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
21                 ↪ powerOf2ToUnaryNumberConverter);
22             var random = new System.Random(0);
23             ulong[] numbers = new ulong[N];
24             ulong[] unaryNumbers = new ulong[N];
25             for (int i = 0; i < N; i++)
26             {
27                 numbers[i] = random.NextUInt64();
28                 unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
29             }
30             var fromUnaryNumberConverterUsingOrOperation = new
31                 ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
32                 ↪ powerOf2ToUnaryNumberConverter);
33             var fromUnaryNumberConverterUsingAddOperation = new
34                 ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
35             for (int i = 0; i < N; i++)
36             {
37                 Assert.Equal(numbers[i],
38                     ↪ fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
39                 Assert.Equal(numbers[i],
40                     ↪ fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
41             }
42         }
43     }
44 }

```

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

```

1  using Xunit;
2  using Platform.Converters;
3  using Platform.Memory;
4  using Platform.Reflection;
5  using Platform.Scopes;
6  using Platform.Data.Numbers.Raw;
7  using Platform.Data.Doublets.Incrementers;
8  using Platform.Data.Doublets.Numbers.Unary;
9  using Platform.Data.Doublets.PropertyOperators;
10 using Platform.Data.Doublets.Sequences.Converters;
11 using Platform.Data.Doublets.Sequences.Indexes;
12 using Platform.Data.Doublets.Sequences.Walkers;
13 using Platform.Data.Doublets.Unicode;
14 using Platform.Data.Doublets.Memory.United.Generic;
15 using Platform.Data.Doublets.CriterionMatchers;
16
17 namespace Platform.Data.Doublets.Tests
18 {
19     public static class UnicodeConvertersTests
20     {
21         [Fact]
22         public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
23         {
24             using (var scope = new TempLinksTestScope())
25             {
26                 var links = scope.Links;
27                 var meaningRoot = links.CreatePoint();
28                 var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
29                 var powerOf2ToUnaryNumberConverter = new
30                     ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
31                 var addressToUnaryNumberConverter = new
32                     ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
33                 var unaryNumberToAddressConverter = new
34                     ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
35                     ↪ powerOf2ToUnaryNumberConverter);
36                 TestCharAndUnicodeSymbolConverters(links, meaningRoot,
37                     ↪ addressToUnaryNumberConverter, unaryNumberToAddressConverter);
38             }
39         }
40     }
41 }

```

```

36 [Fact]
37 public static void CharAndRawNumberUnicodeSymbolConvertersTest()
38 {
39     using (var scope = new Scope<Types<HeapResizableDirectMemory,
40         ↳ UnitedMemoryLinks<ulong>>>())
41     {
42         var links = scope.Use<ILinks<ulong>>();
43         var meaningRoot = links.CreatePoint();
44         var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
45         var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
46         TestCharAndUnicodeSymbolConverters(links, meaningRoot,
47             ↳ addressToRawNumberConverter, rawNumberToAddressConverter);
48     }
49 }
50
51 private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
52     ↳ meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
53     ↳ numberToAddressConverter)
54 {
55     var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
56     var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
57         ↳ addressToNumberConverter, unicodeSymbolMarker);
58     var originalCharacter = 'H';
59     var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
60     var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
61         ↳ unicodeSymbolMarker);
62     var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
63         ↳ numberToAddressConverter, unicodeSymbolCriterionMatcher);
64     var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
65     Assert.Equal(originalCharacter, resultingCharacter);
66 }
67
68 [Fact]
69 public static void StringAndUnicodeSequenceConvertersTest()
70 {
71     using (var scope = new TempLinksTestScope())
72     {
73         var links = scope.Links;
74
75         var itself = links.Constants.Itself;
76
77         var meaningRoot = links.CreatePoint();
78         var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
79         var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
80         var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
81         var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
82         var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
83
84         var powerOf2ToUnaryNumberConverter = new
85             ↳ PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
86         var addressToUnaryNumberConverter = new
87             ↳ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
88         var charToUnicodeSymbolConverter = new
89             ↳ CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
90                 ↳ unicodeSymbolMarker);
91
92         var unaryNumberToAddressConverter = new
93             ↳ UnaryNumberToAddressOrOperationConverter<ulong>(links,
94                 ↳ powerOf2ToUnaryNumberConverter);
95         var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
96         var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
97             ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
98         var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
99             ↳ frequencyPropertyMarker, frequencyMarker);
100        var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
101            ↳ frequencyPropertyOperator, frequencyIncrementer);
102        var linkToItsFrequencyNumberConverter = new
103            ↳ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
104                ↳ unaryNumberToAddressConverter);
105        var sequenceToItsLocalElementLevelsConverter = new
106            ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
107                ↳ linkToItsFrequencyNumberConverter);
108        var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
109            ↳ sequenceToItsLocalElementLevelsConverter);

```

```

90     var stringToUnicodeSequenceConverter = new
91         ↳ StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
92         ↳ index, optimalVariantConverter, unicodeSequenceMarker);
93
94     var originalString = "Hello";
95
96     var unicodeSequenceLink =
97         ↳ stringToUnicodeSequenceConverter.Convert(originalString);
98
99     var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
100         ↳ unicodeSymbolMarker);
101     var unicodeSymbolToCharConverter = new
102         ↳ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
103         ↳ unicodeSymbolCriterionMatcher);
104
105     var unicodeSequenceCriterionMatcher = new TargetMatcher<ulong>(links,
106         ↳ unicodeSequenceMarker);
107
108     var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
109         ↳ unicodeSymbolCriterionMatcher.IsMatched);
110
111     var unicodeSequenceToStringConverter = new
112         ↳ UnicodeSequenceToStringConverter<ulong>(links,
113         ↳ unicodeSequenceCriterionMatcher, sequenceWalker,
114         ↳ unicodeSymbolToCharConverter);
115
116     var resultingString =
117         ↳ unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
118
119     Assert.Equal(originalString, resultingString);
120 }
121 }
122 }
123 }
124 }

```

Index

`./csharp/Platform.Data.Doublets.Tests/GenericLinksTests.cs`, 171
`./csharp/Platform.Data.Doublets.Tests/ILinksExtensionsTests.cs`, 172
`./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs`, 172
`./csharp/Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs`, 172
`./csharp/Platform.Data.Doublets.Tests/ReadSequenceTests.cs`, 176
`./csharp/Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs`, 176
`./csharp/Platform.Data.Doublets.Tests/ScopeTests.cs`, 177
`./csharp/Platform.Data.Doublets.Tests/SequencesTests.cs`, 178
`./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs`, 193
`./csharp/Platform.Data.Doublets.Tests/TempLinksTestScope.cs`, 194
`./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs`, 194
`./csharp/Platform.Data.Doublets.Tests/UInt64LinksTests.cs`, 197
`./csharp/Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs`, 209
`./csharp/Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs`, 210
`./csharp/Platform.Data.Doublets/CriterionMatchers/TargetMatcher.cs`, 1
`./csharp/Platform.Data.Doublets/Decorators/LinksCascadeUniquenessAndUsagesResolver.cs`, 1
`./csharp/Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.cs`, 1
`./csharp/Platform.Data.Doublets/Decorators/LinksDecoratorBase.cs`, 1
`./csharp/Platform.Data.Doublets/Decorators/LinksDisposableDecoratorBase.cs`, 2
`./csharp/Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.cs`, 3
`./csharp/Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.cs`, 3
`./csharp/Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.cs`, 4
`./csharp/Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.cs`, 4
`./csharp/Platform.Data.Doublets/Decorators/LinksUniquenessResolver.cs`, 5
`./csharp/Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs`, 5
`./csharp/Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs`, 5
`./csharp/Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs`, 6
`./csharp/Platform.Data.Doublets/Decorators/UInt64Links.cs`, 6
`./csharp/Platform.Data.Doublets/Decorators/UniLinks.cs`, 7
`./csharp/Platform.Data.Doublets/Doublet.cs`, 12
`./csharp/Platform.Data.Doublets/DoubletComparer.cs`, 13
`./csharp/Platform.Data.Doublets/ILinks.cs`, 13
`./csharp/Platform.Data.Doublets/ILinksExtensions.cs`, 13
`./csharp/Platform.Data.Doublets/ISynchronizedLinks.cs`, 25
`./csharp/Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs`, 25
`./csharp/Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs`, 26
`./csharp/Platform.Data.Doublets/Link.cs`, 26
`./csharp/Platform.Data.Doublets/LinkExtensions.cs`, 29
`./csharp/Platform.Data.Doublets/LinksOperatorBase.cs`, 30
`./csharp/Platform.Data.Doublets/Memory/ILinksListMethods.cs`, 30
`./csharp/Platform.Data.Doublets/Memory/ILinksTreeMethods.cs`, 30
`./csharp/Platform.Data.Doublets/Memory/LinksHeader.cs`, 30
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSizeBalancedTreeMethodsBase.cs`, 31
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSourcesSizeBalancedTreeMethods.cs`, 34
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsSizeBalancedTreeMethods.cs`, 35
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSizeBalancedTreeMethodsBase.cs`, 36
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesSizeBalancedTreeMethods.cs`, 39
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsSizeBalancedTreeMethods.cs`, 40
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinks.cs`, 40
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinksBase.cs`, 42
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/UnusedLinksListMethods.cs`, 51
`./csharp/Platform.Data.Doublets/Memory/Split/RawLinkDataPart.cs`, 52
`./csharp/Platform.Data.Doublets/Memory/Split/RawLinkIndexPart.cs`, 53
`./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksAvlBalancedTreeMethodsBase.cs`, 53
`./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSizeBalancedTreeMethodsBase.cs`, 58
`./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesAvlBalancedTreeMethods.cs`, 61
`./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesSizeBalancedTreeMethods.cs`, 62
`./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsAvlBalancedTreeMethods.cs`, 63
`./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsSizeBalancedTreeMethods.cs`, 64
`./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinks.cs`, 65
`./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinksBase.cs`, 66
`./csharp/Platform.Data.Doublets/Memory/United/Generic/UnusedLinksListMethods.cs`, 73
`./csharp/Platform.Data.Doublets/Memory/United/RawLink.cs`, 74
`./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs`, 75
`./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs`, 77
`./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs`, 78

./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.cs, 79
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs, 80
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs, 81
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnitedMemoryLinks.cs, 82
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnusedLinksListMethods.cs, 84
./csharp/Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs, 84
./csharp/Platform.Data.Doublets/Numbers/Unary/LinkToltsFrequencyNumberConveter.cs, 85
./csharp/Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs, 86
./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs, 86
./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs, 87
./csharp/Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 88
./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 89
./csharp/Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 90
./csharp/Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 91
./csharp/Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 94
./csharp/Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 94
./csharp/Platform.Data.Doublets/Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 96
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs, 96
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs, 97
./csharp/Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 97
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 98
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 98
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 101
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 102
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToltsFrequencyValueConverter.cs, 103
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 103
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 103
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 104
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 105
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 105
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 105
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 106
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 107
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 107
./csharp/Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 107
./csharp/Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 108
./csharp/Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 109
./csharp/Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 109
./csharp/Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 110
./csharp/Platform.Data.Doublets/Sequences/Indexes/Unindex.cs, 111
./csharp/Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 111
./csharp/Platform.Data.Doublets/Sequences/Sequences.cs, 138
./csharp/Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 149
./csharp/Platform.Data.Doublets/Sequences/SequencesOptions.cs, 149
./csharp/Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 152
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 152
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 153
./csharp/Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 155
./csharp/Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 155
./csharp/Platform.Data.Doublets/Stacks/Stack.cs, 156
./csharp/Platform.Data.Doublets/Stacks/StackExtensions.cs, 157
./csharp/Platform.Data.Doublets/SynchronizedLinks.cs, 157
./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs, 158
./csharp/Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 160
./csharp/Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 166
./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 166
./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSymbolsListConverter.cs, 167
./csharp/Platform.Data.Doublets/Unicode/UnicodeMap.cs, 167
./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 170
./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 171