

LinksPlatform's Platform.Data.Doublets Class Library

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

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

1.2 ./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 }
```

1.3 ./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         private ILinks<TLink> _facade;
12
13         public LinksConstants<TLink> Constants { get; }
14
15         public ILinks<TLink> Facade
16         {
17             get => _facade;
18             set
19             {
20                 _facade = value;
21                 if (Links is LinksDecoratorBase<TLink> decorator)
22                 {
23                     decorator.Facade = value;
24                 }
25                 else if (Links is LinksDisposableDecoratorBase<TLink> disposableDecorator)
26                 {
```

```

27         disposableDecorator.Facade = value;
28     }
29 }
30
31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 protected LinksDecoratorBase(ILinks<TLink> links) : base(links)
33 {
34     Constants = links.Constants;
35     Facade = this;
36 }
37
38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 public virtual TLink Count(IList<TLink> restrictions) => Links.Count(restrictions);
40
41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
43     => Links.Each(handler, restrictions);
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 public virtual TLink Create(IList<TLink> restrictions) => Links.Create(restrictions);
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
50     Links.Update(restrictions, substitution);
51
52 [MethodImpl(MethodImplOptions.AggressiveInlining)]
53 public virtual void Delete(IList<TLink> restrictions) => Links.Delete(restrictions);
54 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Decorators
9  {
10     public abstract class LinksDisposableDecoratorBase<TLink> : DisposableBase, ILinks<TLink>
11     {
12         private ILinks<TLink> _facade;
13
14         public LinksConstants<TLink> Constants { get; }
15
16         public ILinks<TLink> Links { get; }
17
18         public ILinks<TLink> Facade
19         {
20             get => _facade;
21             set
22             {
23                 _facade = value;
24                 if (Links is LinksDecoratorBase<TLink> decorator)
25                 {
26                     decorator.Facade = value;
27                 }
28                 else if (Links is LinksDisposableDecoratorBase<TLink> disposableDecorator)
29                 {
30                     disposableDecorator.Facade = value;
31                 }
32             }
33         }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected LinksDisposableDecoratorBase(ILinks<TLink> links)
37         {
38             Links = links;
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);

```

```

48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     public virtual TLink Create(IList<TLink> restrictions) => Links.Create(restrictions);
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
53         ↳ Links.Update(restrictions, substitution);
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     public virtual void Delete(IList<TLink> restrictions) => Links.Delete(restrictions);
57
58     protected override bool AllowMultipleDisposeCalls => true;
59
60     protected override void Dispose(bool manual, bool wasDisposed)
61     {
62         if (!wasDisposed)
63         {
64             Links.DisposeIfPossible();
65         }
66     }
67 }
68 }

```

1.5 ./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             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
20             return Links.Each(handler, restrictions);
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
25         {
26             // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
27             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
28             Links.EnsureInnerReferenceExists(substitution, nameof(substitution));
29             return Links.Update(restrictions, substitution);
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public override void Delete(IList<TLink> restrictions)
34         {
35             var link = restrictions[Constants.IndexPart];
36             Links.EnsureLinkExists(link, nameof(link));
37             Links.Delete(link);
38         }
39     }
40 }

```

1.6 ./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 }

```

```

13     [MethodImpl(MethodImplOptions.AggressiveInlining)]
14     public LinksItselfConstantToSelfReferenceResolver(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 constants = Constants;
20         var itselfConstant = constants.Itself;
21         var indexPartConstant = constants.IndexPart;
22         var sourcePartConstant = constants.SourcePart;
23         var targetPartConstant = constants.TargetPart;
24         var restrictionsCount = restrictions.Count;
25         if (!_equalityComparer.Equals(constants.Any, itselfConstant)
26             && (((restrictionsCount > indexPartConstant) &&
27                 ↪ _equalityComparer.Equals(restrictions[indexPartConstant], itselfConstant))
28                 || ((restrictionsCount > sourcePartConstant) &&
29                     ↪ _equalityComparer.Equals(restrictions[sourcePartConstant], itselfConstant))
30                 || ((restrictionsCount > targetPartConstant) &&
31                     ↪ _equalityComparer.Equals(restrictions[targetPartConstant], itselfConstant))))
32         {
33             // Itself constant is not supported for Each method right now, skipping execution
34             return constants.Continue;
35         }
36         return Links.Each(handler, restrictions);
37     }
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
41     ↪ Links.Update(restrictions, Links.ResolveConstantAsSelfReference(Constants.Itself,
42     ↪ restrictions, substitution));
43 }

```

1.7 ./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            Links.EnsureCreated(substitution[constants.SourcePart],
25            ↪ substitution[constants.TargetPart]);
26            return Links.Update(restrictions, substitution);
27        }
28    }
29 }

```

1.8 ./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(IList<TLink> restrictions)

```

```

15     {
16         var link = Links.Create();
17         return Links.Update(link, link, link);
18     }
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
22     ↪ Links.Update(restrictions, Links.ResolveConstantAsSelfReference(Constants.Null,
23     ↪ restrictions, substitution));
24 }

```

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

```

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

```

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

```

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

```

1.11 ./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             Links.EnsureNoUsages(restrictions[Constants.IndexPart]);
17             return Links.Update(restrictions, substitution);
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public override void Delete(IList<TLink> restrictions)
22         {
23             var link = restrictions[Constants.IndexPart];
24             Links.EnsureNoUsages(link);
25             Links.Delete(link);
26         }
27     }
28 }
```

1.12 ./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             Links.EnforceResetValues(linkIndex);
18             Links.Delete(linkIndex);
19         }
20     }
21 }
```

1.13 ./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     /// Представляет объект для работы с базой данных (файлом) в формате Links (массива связей).
10     /// </summary>
11     /// <remarks>
12     /// Возможные оптимизации:
13     /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
14     ///     + меньше объём БД
15     ///     - меньше производительность
16     ///     - больше ограничение на количество связей в БД)
17     /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
18     ///     + меньше объём БД
19     ///     - больше сложность
20     ///
21     /// Текущее теоретическое ограничение на индекс связи, из-за использования 5 бит в размере
22     ///     ↳ поддеревьев для AVL баланса и флагов нитей: 2 в степени(64 минус 5 равно 59 ) равно 576
23     ///     ↳ 460 752 303 423 488
24     /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
25     ///     ↳ (битовыми строками) - вариант матрицы (выстраиваемой лениво).
```

```

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

```

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

```

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

```

```

18 internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
19 {
20     private static readonly EqualityComparer<TLink> _equalityComparer =
21         ↪ EqualityComparer<TLink>.Default;
22
23     public UniLinks(ILinks<TLink> links) : base(links) { }
24
25     private struct Transition
26     {
27         public IList<TLink> Before;
28         public IList<TLink> After;
29
30         public Transition(IList<TLink> before, IList<TLink> after)
31         {
32             Before = before;
33             After = after;
34         }
35     }
36
37     //public static readonly TLink NullConstant = Use<LinksConstants<TLink>>.Single.Null;
38     //public static readonly IReadOnlyList<TLink> NullLink = new
39     ↪ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
40     ↪ });
41
42     // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
43     ↪ (Links-Expression)
44     public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
45     ↪ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
46     ↪ substitutedHandler)
47     {
48         ///List<Transition> transitions = null;
49         ///if (!restriction.IsNullOrEmpty())
50         ///{
51         ///    // Есть причина делать проход (чтение)
52         ///    if (matchedHandler != null)
53         ///    {
54         ///        if (!substitution.IsNullOrEmpty())
55         ///        {
56         ///            // restriction => { 0, 0, 0 } | { 0 } // Create
57         ///            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
58         ↪ Create / Update
59         ///            // substitution => { 0, 0, 0 } | { 0 } // Delete
60         ///            transitions = new List<Transition>();
61         ///            if (Equals(substitution[Constants.IndexPart], Constants.Null))
62         ///            {
63         ///                // If index is Null, that means we always ignore every other
64         ↪ value (they are also Null by definition)
65         ///                var matchDecision = matchedHandler(, NullLink);
66         ///                if (Equals(matchDecision, Constants.Break))
67         ///                    return false;
68         ///                if (!Equals(matchDecision, Constants.Skip))
69         ///                    transitions.Add(new Transition(matchedLink, newValue));
70         ///            }
71         ///            else
72         ///            {
73         ///                Func<T, bool> handler;
74         ///                handler = link =>
75         ///                {
76         ///                    var matchedLink = Memory.GetLinkValue(link);
77         ///                    var newValue = Memory.GetLinkValue(link);
78         ///                    newValue[Constants.IndexPart] = Constants.Itself;
79         ///                    newValue[Constants.SourcePart] =
80         ↪ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
81         ↪ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
82         ///                    newValue[Constants.TargetPart] =
83         ↪ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
84         ↪ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
85         ///                    var matchDecision = matchedHandler(matchedLink, newValue);
86         ///                    if (Equals(matchDecision, Constants.Break))
87         ///                        return false;
88         ///                    if (!Equals(matchDecision, Constants.Skip))
89         ///                        transitions.Add(new Transition(matchedLink, newValue));
90         ///                    return true;
91         ///                };
92         ///            if (!Memory.Each(handler, restriction))
93         ///                return Constants.Break;
94         ///        }
95     }

```



```

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    //    {
151    //        matchedLinks = new List<IList<T>>>();
152    //        Func<T, bool> handler = link =>
153    //        {
154    //            var matchedLink = Memory.GetLinkValue(link);
155    //            var matchDecision = matchedHandler(matchedLink);
156    //            if (Equals(matchDecision, Constants.Break))
157    //                return false;
158    //            if (!Equals(matchDecision, Constants.Skip))
159    //                matchedLinks.Add(matchedLink);
160    //            return true;
161    //        };

```

```

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     // }
194     // }
195     // }
196     // }
197     // }
198     // }
199     // }
200     // }
201     // }
202     // }
203     // }
204     // }
205     // }
206     // }
207     // }
208     // }
209     // }
210     // }
211     // }
212     // }
213     // }
214     // }
215     // }
216     // }
217     // }
218     // }
219     // }
220     // }
221     // }
222     // }
223     // }
224     // }
225     // }
226     // }
227     // }
228     // }
229     // }
230     // }
231     // }
232     // }
233     // }
234     // }
235     // }
236     // }
237     // }
238     // }
239     // }
240     // }
241     // }
242     // }
243     // }
244     // }
245     // }
246     // }
247     // }
248     // }
249     // }
250     // }
251     // }
252     // }
253     // }
254     // }
255     // }
256     // }
257     // }
258     // }
259     // }
260     // }
261     // }
262     // }
263     // }
264     // }
265     // }
266     // }
267     // }
268     // }
269     // }
270     // }
271     // }
272     // }
273     // }
274     // }
275     // }
276     // }
277     // }
278     // }
279     // }
280     // }
281     // }
282     // }
283     // }
284     // }
285     // }
286     // }
287     // }
288     // }
289     // }
290     // }
291     // }
292     // }
293     // }
294     // }
295     // }
296     // }
297     // }
298     // }
299     // }
300     // }
301     // }
302     // }
303     // }
304     // }
305     // }
306     // }
307     // }
308     // }
309     // }
310     // }
311     // }
312     // }
313     // }
314     // }
315     // }
316     // }
317     // }
318     // }
319     // }
320     // }
321     // }
322     // }
323     // }
324     // }
325     // }
326     // }
327     // }
328     // }
329     // }
330     // }
331     // }
332     // }
333     // }
334     // }
335     // }
336     // }
337     // }
338     // }
339     // }
340     // }
341     // }
342     // }
343     // }
344     // }
345     // }
346     // }
347     // }
348     // }
349     // }
350     // }
351     // }
352     // }
353     // }
354     // }
355     // }
356     // }
357     // }
358     // }
359     // }
360     // }
361     // }
362     // }
363     // }
364     // }
365     // }
366     // }
367     // }
368     // }
369     // }
370     // }
371     // }
372     // }
373     // }
374     // }
375     // }
376     // }
377     // }
378     // }
379     // }
380     // }
381     // }
382     // }
383     // }
384     // }
385     // }
386     // }
387     // }
388     // }
389     // }
390     // }
391     // }
392     // }
393     // }
394     // }
395     // }
396     // }
397     // }
398     // }
399     // }
400     // }
401     // }
402     // }
403     // }
404     // }
405     // }
406     // }
407     // }
408     // }
409     // }
410     // }
411     // }
412     // }
413     // }
414     // }
415     // }
416     // }
417     // }
418     // }
419     // }
420     // }
421     // }
422     // }
423     // }
424     // }
425     // }
426     // }
427     // }
428     // }
429     // }
430     // }
431     // }
432     // }
433     // }
434     // }
435     // }
436     // }
437     // }
438     // }
439     // }
440     // }
441     // }
442     // }
443     // }
444     // }
445     // }
446     // }
447     // }
448     // }
449     // }
450     // }
451     // }
452     // }
453     // }
454     // }
455     // }
456     // }
457     // }
458     // }
459     // }
460     // }
461     // }
462     // }
463     // }
464     // }
465     // }
466     // }
467     // }
468     // }
469     // }
470     // }
471     // }
472     // }
473     // }
474     // }
475     // }
476     // }
477     // }
478     // }
479     // }
480     // }
481     // }
482     // }
483     // }
484     // }
485     // }
486     // }
487     // }
488     // }
489     // }
490     // }
491     // }
492     // }
493     // }
494     // }
495     // }
496     // }
497     // }
498     // }
499     // }
500     // }
501     // }
502     // }
503     // }
504     // }
505     // }
506     // }
507     // }
508     // }
509     // }
510     // }
511     // }
512     // }
513     // }
514     // }
515     // }
516     // }
517     // }
518     // }
519     // }
520     // }
521     // }
522     // }
523     // }
524     // }
525     // }
526     // }
527     // }
528     // }
529     // }
530     // }
531     // }
532     // }
533     // }
534     // }
535     // }
536     // }
537     // }
538     // }
539     // }
540     // }
541     // }
542     // }
543     // }
544     // }
545     // }
546     // }
547     // }
548     // }
549     // }
550     // }
551     // }
552     // }
553     // }
554     // }
555     // }
556     // }
557     // }
558     // }
559     // }
560     // }
561     // }
562     // }
563     // }
564     // }
565     // }
566     // }
567     // }
568     // }
569     // }
570     // }
571     // }
572     // }
573     // }
574     // }
575     // }
576     // }
577     // }
578     // }
579     // }
580     // }
581     // }
582     // }
583     // }
584     // }
585     // }
586     // }
587     // }
588     // }
589     // }
590     // }
591     // }
592     // }
593     // }
594     // }
595     // }
596     // }
597     // }
598     // }
599     // }
600     // }
601     // }
602     // }
603     // }
604     // }
605     // }
606     // }
607     // }
608     // }
609     // }
610     // }
611     // }
612     // }
613     // }
614     // }
615     // }
616     // }
617     // }
618     // }
619     // }
620     // }
621     // }
622     // }
623     // }
624     // }
625     // }
626     // }
627     // }
628     // }
629     // }
630     // }
631     // }
632     // }
633     // }
634     // }
635     // }
636     // }
637     // }
638     // }
639     // }
640     // }
641     // }
642     // }
643     // }
644     // }
645     // }
646     // }
647     // }
648     // }
649     // }
650     // }
651     // }
652     // }
653     // }
654     // }
655     // }
656     // }
657     // }
658     // }
659     // }
660     // }
661     // }
662     // }
663     // }
664     // }
665     // }
666     // }
667     // }
668     // }
669     // }
670     // }
671     // }
672     // }
673     // }
674     // }
675     // }
676     // }
677     // }
678     // }
679     // }
680     // }
681     // }
682     // }
683     // }
684     // }
685     // }
686     // }
687     // }
688     // }
689     // }
690     // }
691     // }
692     // }
693     // }
694     // }
695     // }
696     // }
697     // }
698     // }
699     // }
700     // }
701     // }
702     // }
703     // }
704     // }
705     // }
706     // }
707     // }
708     // }
709     // }
710     // }
711     // }
712     // }
713     // }
714     // }
715     // }
716     // }
717     // }
718     // }
719     // }
720     // }
721     // }
722     // }
723     // }
724     // }
725     // }
726     // }
727     // }
728     // }
729     // }
730     // }
731     // }
732     // }
733     // }
734     // }
735     // }
736     // }
737     // }
738     // }
739     // }
740     // }
741     // }
742     // }
743     // }
744     // }
745     // }
746     // }
747     // }
748     // }
749     // }
750     // }
751     // }
752     // }
753     // }
754     // }
755     // }
756     // }
757     // }
758     // }
759     // }
760     // }
761     // }
762     // }
763     // }
764     // }
765     // }
766     // }
767     // }
768     // }
769     // }
770     // }
771     // }
772     // }
773     // }
774     // }
775     // }
776     // }
777     // }
778     // }
779     // }
780     // }
781     // }
782     // }
783     // }
784     // }
785     // }
786     // }
787     // }
788     // }
789     // }
790     // }
791     // }
792     // }
793     // }
794     // }
795     // }
796     // }
797     // }
798     // }
799     // }
800     // }
801     // }
802     // }
803     // }
804     // }
805     // }
806     // }
807     // }
808     // }
809     // }
810     // }
811     // }
812     // }
813     // }
814     // }
815     // }
816     // }
817     // }
818     // }
819     // }
820     // }
821     // }
822     // }
823     // }
824     // }
825     // }
826     // }
827     // }
828     // }
829     // }
830     // }
831     // }
832     // }
833     // }
834     // }
835     // }
836     // }
837     // }
838     // }
839     // }
840     // }
841     // }
842     // }
843     // }
844     // }
845     // }
846     // }
847     // }
848     // }
849     // }
850     // }
851     // }
852     // }
853     // }
854     // }
855     // }
856     // }
857     // }
858     // }
859     // }
860     // }
861     // }
862     // }
863     // }
864     // }
865     // }
866     // }
867     // }
868     // }
869     // }
870     // }
871     // }
872     // }
873     // }
874     // }
875     // }
876     // }
877     // }
878     // }
879     // }
880     // }
881     // }
882     // }
883     // }
884     // }
885     // }
886     // }
887     // }
888     // }
889     // }
890     // }
891     // }
892     // }
893     // }
894     // }
895     // }
896     // }
897     // }
898     // }
899     // }
900     // }
901     // }
902     // }
903     // }
904     // }
905     // }
906     // }
907     // }
908     // }
909     // }
910     // }
911     // }
912     // }
913     // }
914     // }
915     // }
916     // }
917     // }
918     // }
919     // }
920     // }
921     // }
922     // }
923     // }
924     // }
925     // }
926     // }
927     // }
928     // }
929     // }
930     // }
931     // }
932     // }
933     // }
934     // }
935     // }
936     // }
937     // }
938     // }
939     // }
940     // }
941     // }
942     // }
943     // }
944     // }
945     // }
946     // }
947     // }
948     // }
949     // }
950     // }
951     // }
952     // }
953     // }
954     // }
955     // }
956     // }
957     // }
958     // }
959     // }
960     // }
961     // }
962     // }
963     // }
964     // }
965     // }
966     // }
967     // }
968     // }
969     // }
970     // }
971     // }
972     // }
973     // }
974     // }
975     // }
976     // }
977     // }
978     // }
979     // }
980     // }
981     // }
982     // }
983     // }
984     // }
985     // }
986     // }
987     // }
988     // }
989     // }
990     // }
991     // }
992     // }
993     // }
994     // }
995     // }
996     // }
997     // }
998     // }
999     // }
1000    // }

```

```

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

```

```

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

```

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

```

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

```

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

```

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

```

1.17 ./Platform.Data.Doublets/Hybrid.cs

```

1  using System;
2  using System.Reflection;
3  using System.Reflection.Emit;
4  using Platform.Reflection;
5  using Platform.Converters;
6  using Platform.Exceptions;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets
11 {
12     public struct Hybrid<T>
13     {
14         private static readonly Func<object, T> _absAndConvert;
15         private static readonly Func<object, T> _absAndNegateAndConvert;
16
17         static Hybrid()
18         {
19             _absAndConvert = DelegateHelpers.Compile<Func<object, T>>(emitter =>
20             {
21                 Ensure.Always.IsUnsignedInteger<T>();
22                 emitter.LoadArgument(0);
23                 var signedVersion = NumericType<T>.SignedVersion;
24                 var signedVersionField =
25                     ↳ typeof(NumericType<T>).GetTypeInfo().GetField("SignedVersion",
26                     ↳ BindingFlags.Static | BindingFlags.Public);
27                 //emitter.LoadField(signedVersionField);
28                 emitter.Emit(OpCodes.Ldsfld, signedVersionField);
29                 var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
30                     ↳ Types<object, Type>.Array);
31                 emitter.Call(changeTypeMethod);
32                 emitter.UnboxValue(signedVersion);
33                 var absMethod = typeof(Math).GetTypeInfo().GetMethod("Abs", new[] {
34                     ↳ signedVersion });
35                 emitter.Call(absMethod);
36                 var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
37                     ↳ signedVersion });
38                 emitter.Call(unsignedMethod);
39                 emitter.Return();
40             });
41             _absAndNegateAndConvert = DelegateHelpers.Compile<Func<object, T>>(emitter =>
42             {
43                 Ensure.Always.IsUnsignedInteger<T>();
44                 emitter.LoadArgument(0);
45                 var signedVersion = NumericType<T>.SignedVersion;
46                 var signedVersionField =
47                     ↳ typeof(NumericType<T>).GetTypeInfo().GetField("SignedVersion",
48                     ↳ BindingFlags.Static | BindingFlags.Public);
49                 //emitter.LoadField(signedVersionField);
50                 emitter.Emit(OpCodes.Ldsfld, signedVersionField);
51                 var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
52                     ↳ Types<object, Type>.Array);
53                 emitter.Call(changeTypeMethod);
54                 emitter.UnboxValue(signedVersion);
55                 var absMethod = typeof(Math).GetTypeInfo().GetMethod("Abs", new[] {
56                     ↳ signedVersion });
57                 emitter.Call(absMethod);
58                 var negateMethod = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate",
59                     ↳ ").MakeGenericMethod(signedVersion);
60                 emitter.Call(negateMethod);
61                 var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
62                     ↳ signedVersion });
63                 emitter.Call(unsignedMethod);
64                 emitter.Return();
65             });
66         }
67
68         public readonly T Value;
69         public bool IsNothing => Convert.ToInt64(To.Signed(Value)) == 0;
70         public bool IsInternal => Convert.ToInt64(To.Signed(Value)) > 0;
71         public bool IsExternal => Convert.ToInt64(To.Signed(Value)) < 0;
72         public long AbsoluteValue =>
73             ↳ Platform.Numbers.Math.Abs(Convert.ToInt64(To.Signed(Value)));
74     }
75 }

```

```

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

```

1.18 ./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 ./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.Numbers;
9  using Platform.Random;
10 using Platform.Setters;
11 using Platform.Data.Exceptions;
12 using Platform.Data.Doublets.Decorators;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets
17 {
18     public static class ILinksExtensions
19     {
20         public static void RunRandomCreations<TLink>(this ILinks<TLink> links, long
21             ↳ amountOfCreations)
22         {
23             for (long i = 0; i < amountOfCreations; i++)
24             {
25                 var linksAddressRange = new Range<ulong>(0, (Integer<TLink>)links.Count());
26                 Integer<TLink> source = RandomHelpers.Default.NextUInt64(linksAddressRange);
27                 Integer<TLink> target = RandomHelpers.Default.NextUInt64(linksAddressRange);
28                 links.CreateAndUpdate(source, target);
29             }
30
31             public static void RunRandomSearches<TLink>(this ILinks<TLink> links, long
32                 ↳ amountOfSearches)
33             {
34                 for (long i = 0; i < amountOfSearches; i++)
35                 {
36                     var linkAddressRange = new Range<ulong>(1, (Integer<TLink>)links.Count());
37                     Integer<TLink> source = RandomHelpers.Default.NextUInt64(linkAddressRange);
38                     Integer<TLink> target = RandomHelpers.Default.NextUInt64(linkAddressRange);
39                     links.SearchOrDefault(source, target);
40                 }
41
42                 public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, long
43                     ↳ amountOfDeletions)
44                 {
45                     var min = (ulong)amountOfDeletions > (Integer<TLink>)links.Count() ? 1 :
46                         ↳ (Integer<TLink>)links.Count() - (ulong)amountOfDeletions;
47                     for (long i = 0; i < amountOfDeletions; i++)
48                     {
49                         var linksAddressRange = new Range<ulong>(min, (Integer<TLink>)links.Count());
50                         Integer<TLink> link = RandomHelpers.Default.NextUInt64(linksAddressRange);
51                         links.Delete(link);
52                         if ((Integer<TLink>)links.Count() < min)
53                         {
54                             break;
55                         }
56                     }
57
58                     public static void Delete<TLink>(this ILinks<TLink> links, TLink linkToDelete) =>
59                         ↳ links.Delete(new LinkAddress<TLink>(linkToDelete));
60
61                     /// <remarks>
62                     /// TODO: Возможно есть очень простой способ это сделать.
63                     /// (Например просто удалить файл, или изменить его размер таким образом,
```

```

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

```



```

132         return false;
133     }
134     current = next;
135 }
136 return true;
137 }
138
139 /// <remarks>
140 /// Может потребовать дополнительного стека для PathElement's при использовании
141   ↳ SequenceWalker.
142 /// </remarks>
143 public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink root, params int[]
144   ↳ path)
145 {
146     links.EnsureLinkExists(root, "root");
147     var currentLink = root;
148     for (var i = 0; i < path.Length; i++)
149     {
150         currentLink = links.GetLink(currentLink)[path[i]];
151     }
152     return currentLink;
153 }
154
155 public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
156   ↳ links, TLink root, ulong size, ulong index)
157 {
158     var constants = links.Constants;
159     var source = constants.SourcePart;
160     var target = constants.TargetPart;
161     if (!Platform.Numbers.Math.IsPowerOfTwo(size))
162     {
163         throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
164           ↳ than powers of two are not supported.");
165     }
166     var path = new BitArray(BitConverter.GetBytes(index));
167     var length = Bit.GetLowestPosition(size);
168     links.EnsureLinkExists(root, "root");
169     var currentLink = root;
170     for (var i = length - 1; i >= 0; i--)
171     {
172         currentLink = links.GetLink(currentLink)[path[i] ? target : source];
173     }
174     return currentLink;
175 }
176
177 #endregion
178
179 /// <summary>
180 /// Возвращает индекс указанной связи.
181 /// </summary>
182 /// <param name="links">Хранилище связей.</param>
183 /// <param name="link">Связь представленная списком, состоящим из её адреса и
184   ↳ содержимого.</param>
185 /// <returns>Индекс начальной связи для указанной связи.</returns>
186 [MethodImpl(MethodImplOptions.AggressiveInlining)]
187 public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
188   ↳ link[links.Constants.IndexPart];
189
190 /// <summary>
191 /// Возвращает индекс начальной (Source) связи для указанной связи.
192 /// </summary>
193 /// <param name="links">Хранилище связей.</param>
194 /// <param name="link">Индекс связи.</param>
195 /// <returns>Индекс начальной связи для указанной связи.</returns>
196 [MethodImpl(MethodImplOptions.AggressiveInlining)]
197 public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
198   ↳ links.GetLink(link)[links.Constants.SourcePart];
199
200 /// <summary>
201 /// Возвращает индекс начальной (Source) связи для указанной связи.
202 /// </summary>
203 /// <param name="links">Хранилище связей.</param>
204 /// <param name="link">Связь представленная списком, состоящим из её адреса и
205   ↳ содержимого.</param>
206 /// <returns>Индекс начальной связи для указанной связи.</returns>
207 [MethodImpl(MethodImplOptions.AggressiveInlining)]
208 public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
209   ↳ link[links.Constants.SourcePart];

```

```

201
202 /// <summary>
203 /// Возвращает индекс конечной (Target) связи для указанной связи.
204 /// </summary>
205 /// <param name="links">Хранилище связей.</param>
206 /// <param name="link">Индекс связи.</param>
207 /// <returns>Индекс конечной связи для указанной связи.</returns>
208 [MethodImpl(MethodImplOptions.AggressiveInlining)]
209 public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
210     ↪ links.GetLink(link)[links.Constants.TargetPart];
211
212 /// <summary>
213 /// Возвращает индекс конечной (Target) связи для указанной связи.
214 /// </summary>
215 /// <param name="links">Хранилище связей.</param>
216 /// <param name="link">Связь представленная списком, состоящим из её адреса и
217     ↪ содержимого.</param>
218 /// <returns>Индекс конечной связи для указанной связи.</returns>
219 [MethodImpl(MethodImplOptions.AggressiveInlining)]
220 public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
221     ↪ link[links.Constants.TargetPart];
222
223 /// <summary>
224 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
225     ↪ (handler) для каждой подходящей связи.
226 /// </summary>
227 /// <param name="links">Хранилище связей.</param>
228 /// <param name="handler">Обработчик каждой подходящей связи.</param>
229 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
230     ↪ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
231     ↪ Any - отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
232 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
233     ↪ случае.</returns>
234 [MethodImpl(MethodImplOptions.AggressiveInlining)]
235 public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
236     ↪ handler, params TLink[] restrictions)
237     => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
238     ↪ links.Constants.Continue);
239
240 /// <summary>
241 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
242     ↪ (handler) для каждой подходящей связи.
243 /// </summary>
244 /// <param name="links">Хранилище связей.</param>
245 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
246     ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
247     ↪ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
248 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
249     ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
250     ↪ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
251 /// <param name="handler">Обработчик каждой подходящей связи.</param>
252 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
253     ↪ случае.</returns>
254 [MethodImpl(MethodImplOptions.AggressiveInlining)]
255 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
256     ↪ Func<TLink, bool> handler)
257 {
258     var constants = links.Constants;
259     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
260     ↪ constants.Break, constants.Any, source, target);
261 }
262
263 /// <summary>
264 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
265     ↪ (handler) для каждой подходящей связи.
266 /// </summary>
267 /// <param name="links">Хранилище связей.</param>
268 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
269     ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
270     ↪ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
271 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
272     ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
273     ↪ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
274 /// <param name="handler">Обработчик каждой подходящей связи.</param>
275 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
276     ↪ случае.</returns>

```

```

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

```

```

322 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
    ↳ IList<TLink> restrictions, string argumentName)
323 {
324     for (int i = 0; i < restrictions.Count; i++)
325     {
326         links.EnsureInnerReferenceExists(restrictions[i], argumentName);
327     }
328 }
329
330 [MethodImpl(MethodImplOptions.AggressiveInlining)]
331 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
    ↳ restrictions)
332 {
333     var equalityComparer = EqualityComparer<TLink>.Default;
334     var any = links.Constants.Any;
335     for (var i = 0; i < restrictions.Count; i++)
336     {
337         if (!equalityComparer.Equals(restrictions[i], any) &&
            ↳ !links.Exists(restrictions[i]))
338         {
339             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                ↳ $"{sequence[{i}]"}");
340         }
341     }
342 }
343
344 [MethodImpl(MethodImplOptions.AggressiveInlining)]
345 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
    ↳ string argumentName)
346 {
347     var equalityComparer = EqualityComparer<TLink>.Default;
348     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
349     {
350         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
351     }
352 }
353
354 [MethodImpl(MethodImplOptions.AggressiveInlining)]
355 public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
    ↳ link, string argumentName)
356 {
357     var equalityComparer = EqualityComparer<TLink>.Default;
358     if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
359     {
360         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
361     }
362 }
363
364 /// <param name="links">Хранилище связей.</param>
365 [MethodImpl(MethodImplOptions.AggressiveInlining)]
366 public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
    ↳ TLink target)
367 {
368     if (links.Exists(source, target))
369     {
370         throw new LinkWithSameValueAlreadyExistsException();
371     }
372 }
373
374 /// <param name="links">Хранилище связей.</param>
375 public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
376 {
377     if (links.HasUsages(link))
378     {
379         throw new ArgumentLinkHasDependenciesException<TLink>(link);
380     }
381 }
382
383 /// <param name="links">Хранилище связей.</param>
384 public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ addresses) => links.EnsureCreated(links.Create, addresses);
385
386 /// <param name="links">Хранилище связей.</param>
387 public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
388
389 /// <param name="links">Хранилище связей.</param>

```

```

390 public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
391   ↪ params TLink[] addresses)
392 {
393     var constants = links.Constants;
394     var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
395   ↪ !links.Exists(x)));
396     if (nonExistentAddresses.Count > 0)
397     {
398         var max = nonExistentAddresses.Max();
399         max = (Integer<TLink>)System.Math.Min((ulong)(Integer<TLink>)max,
400   ↪ (ulong)(Integer<TLink>)constants.InternalReferencesRange.Maximum);
401         var createdLinks = new List<TLink>();
402         var equalityComparer = EqualityComparer<TLink>.Default;
403         TLink createdLink = creator();
404         while (!equalityComparer.Equals(createdLink, max))
405         {
406             createdLinks.Add(createdLink);
407         }
408         for (var i = 0; i < createdLinks.Count; i++)
409         {
410             if (!nonExistentAddresses.Contains(createdLinks[i]))
411             {
412                 links.Delete(createdLinks[i]);
413             }
414         }
415     }
416 }
417
418 #endregion
419
420 /// <param name="links">Хранилище связей.</param>
421 public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
422 {
423     var constants = links.Constants;
424     var values = links.GetLink(link);
425     TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link,
426   ↪ constants.Any));
427     var equalityComparer = EqualityComparer<TLink>.Default;
428     if (equalityComparer.Equals(values[constants.SourcePart], link))
429     {
430         usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
431     }
432     TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
433   ↪ link));
434     if (equalityComparer.Equals(values[constants.TargetPart], link))
435     {
436         usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
437     }
438     return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
439 }
440
441 /// <param name="links">Хранилище связей.</param>
442 [MethodImpl(MethodImplOptions.AggressiveInlining)]
443 public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
444   ↪ Comparer<TLink>.Default.Compare(links.CountUsages(link), Integer<TLink>.Zero) > 0;
445
446 /// <param name="links">Хранилище связей.</param>
447 [MethodImpl(MethodImplOptions.AggressiveInlining)]
448 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
449   ↪ TLink target)
450 {
451     var constants = links.Constants;
452     var values = links.GetLink(link);
453     var equalityComparer = EqualityComparer<TLink>.Default;
454     return equalityComparer.Equals(values[constants.SourcePart], source) &&
455   ↪ equalityComparer.Equals(values[constants.TargetPart], target);
456 }
457
458 /// <summary>
459 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
460 /// </summary>
461 /// <param name="links">Хранилище связей.</param>
462 /// <param name="source">Индекс связи, которая является началом для искомой
463   ↪ связи.</param>
464 /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
465 /// <returns>Индекс искомой связи с указанными Source (началом) и Target
466   ↪ (концом).</returns>
467 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

458 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↪ target)
459 {
460     var constants = links.Constants;
461     var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break, default);
462     links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
463     return setter.Result;
464 }
465
466 /// <param name="links">Хранилище связей.</param>
467 [MethodImpl(MethodImplOptions.AggressiveInlining)]
468 public static TLink Create<TLink>(this ILinks<TLink> links) => links.Create(null);
469
470 /// <param name="links">Хранилище связей.</param>
471 [MethodImpl(MethodImplOptions.AggressiveInlining)]
472 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
473 {
474     var link = links.Create();
475     return links.Update(link, link, link);
476 }
477
478 /// <param name="links">Хранилище связей.</param>
479 [MethodImpl(MethodImplOptions.AggressiveInlining)]
480 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↪ target) => links.Update(links.Create(), source, target);
481
482 /// <summary>
483 /// Обновляет связь с указанными началом (Source) и концом (Target)
484 /// на связь с указанными началом (NewSource) и концом (NewTarget).
485 /// </summary>
486 /// <param name="links">Хранилище связей.</param>
487 /// <param name="link">Индекс обновляемой связи.</param>
488 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    ↪ выполняется обновление.</param>
489 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    ↪ выполняется обновление.</param>
490 /// <returns>Индекс обновлённой связи.</returns>
491 [MethodImpl(MethodImplOptions.AggressiveInlining)]
492 public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
    ↪ TLink newTarget) => links.Update(new LinkAddress<TLink>(link), new Link<TLink>(link,
    ↪ newSource, newTarget));
493
494 /// <summary>
495 /// Обновляет связь с указанными началом (Source) и концом (Target)
496 /// на связь с указанными началом (NewSource) и концом (NewTarget).
497 /// </summary>
498 /// <param name="links">Хранилище связей.</param>
499 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
    ↪ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
    ↪ Itself - требование установить ссылку на себя, 1..∞ конкретный адрес другой
    ↪ связи.</param>
500 /// <returns>Индекс обновлённой связи.</returns>
501 [MethodImpl(MethodImplOptions.AggressiveInlining)]
502 public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
503 {
504     if (restrictions.Length == 2)
505     {
506         return links.MergeAndDelete(restrictions[0], restrictions[1]);
507     }
508     if (restrictions.Length == 4)
509     {
510         return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
            ↪ restrictions[2], restrictions[3]);
511     }
512     else
513     {
514         return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
515     }
516 }
517
518 [MethodImpl(MethodImplOptions.AggressiveInlining)]
519 public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
    ↪ links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
520 {
521     var equalityComparer = EqualityComparer<TLink>.Default;
522     var constants = links.Constants;
523     var restrictionsIndex = restrictions[constants.IndexPart];
524     var substitutionIndex = substitution[constants.IndexPart];

```

```

525     if (equalityComparer.Equals(substitutionIndex, default))
526     {
527         substitutionIndex = restrictionsIndex;
528     }
529     var source = substitution[constants.SourcePart];
530     var target = substitution[constants.TargetPart];
531     source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
532     target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
533     return new Link<TLink>(substitutionIndex, source, target);
534 }
535
536 /// <summary>
537 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
538   ↳ с указанными Source (началом) и Target (концом).
539 /// </summary>
540 /// <param name="links">Хранилище связей.</param>
541 /// <param name="source">Индекс связи, которая является началом на создаваемой
542   ↳ связи.</param>
543 /// <param name="target">Индекс связи, которая является концом для создаваемой
544   ↳ связи.</param>
545 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
546 [MethodImpl(MethodImplOptions.AggressiveInlining)]
547 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
548   ↳ target)
549 {
550     var link = links.SearchOrDefault(source, target);
551     if (EqualityComparer<TLink>.Default.Equals(link, default))
552     {
553         link = links.CreateAndUpdate(source, target);
554     }
555     return link;
556 }
557
558 /// <summary>
559 /// Обновляет связь с указанными началом (Source) и концом (Target)
560   ↳ на связь с указанными началом (NewSource) и концом (NewTarget).
561 /// </summary>
562 /// <param name="links">Хранилище связей.</param>
563 /// <param name="source">Индекс связи, которая является началом обновляемой
564   ↳ связи.</param>
565 /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
566 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
567   ↳ выполняется обновление.</param>
568 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
569   ↳ выполняется обновление.</param>
570 /// <returns>Индекс обновлённой связи.</returns>
571 [MethodImpl(MethodImplOptions.AggressiveInlining)]
572 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
573   ↳ TLink target, TLink newSource, TLink newTarget)
574 {
575     var equalityComparer = EqualityComparer<TLink>.Default;
576     var link = links.SearchOrDefault(source, target);
577     if (equalityComparer.Equals(link, default))
578     {
579         return links.CreateAndUpdate(newSource, newTarget);
580     }
581     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
582   ↳ target))
583     {
584         return link;
585     }
586     return links.Update(link, newSource, newTarget);
587 }
588
589 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
590 /// <param name="links">Хранилище связей.</param>
591 /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
592 /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
593 [MethodImpl(MethodImplOptions.AggressiveInlining)]
594 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
595   ↳ target)
596 {
597     var link = links.SearchOrDefault(source, target);
598     if (!EqualityComparer<TLink>.Default.Equals(link, default))
599     {
600         links.Delete(link);
601         return link;
602     }

```

```

592     }
593     return default;
594 }
595
596 /// <summary>Удаляет несколько связей.</summary>
597 /// <param name="links">Хранилище связей.</param>
598 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
599 [MethodImpl(MethodImplOptions.AggressiveInlining)]
600 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
601 {
602     for (int i = 0; i < deletedLinks.Count; i++)
603     {
604         links.Delete(deletedLinks[i]);
605     }
606 }
607
608 /// <remarks>Before execution of this method ensure that deleted link is detached (all
609 ↪ values - source and target are reset to null) or it might enter into infinite
610 ↪ recursion.</remarks>
611 public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
612 {
613     var anyConstant = links.Constants.Any;
614     var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
615     links.DeleteByQuery(usagesAsSourceQuery);
616     var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
617     links.DeleteByQuery(usagesAsTargetQuery);
618 }
619
620 public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
621 {
622     var count = (Integer<TLink>)links.Count(query);
623     if (count > 0)
624     {
625         var queryResult = new TLink[count];
626         var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
627             ↪ links.Constants.Continue);
628         links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
629         for (var i = (long)count - 1; i >= 0; i--)
630         {
631             links.Delete(queryResult[i]);
632         }
633     }
634 }
635
636 // TODO: Move to Platform.Data
637 public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
638 {
639     var nullConstant = links.Constants.Null;
640     var equalityComparer = EqualityComparer<TLink>.Default;
641     var link = links.GetLink(linkIndex);
642     for (int i = 1; i < link.Count; i++)
643     {
644         if (!equalityComparer.Equals(link[i], nullConstant))
645         {
646             return false;
647         }
648     }
649     return true;
650 }
651
652 // TODO: Create a universal version of this method in Platform.Data (with using of for
653 ↪ loop)
654 public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
655 {
656     var nullConstant = links.Constants.Null;
657     var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
658     links.Update(updateRequest);
659 }
660
661 // TODO: Create a universal version of this method in Platform.Data (with using of for
662 ↪ loop)
663 public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
664 {
665     if (!links.AreValuesReset(linkIndex))
666     {
667         links.ResetValues(linkIndex);
668     }
669 }

```



```

665
666 /// <summary>
667 /// Merging two usages graphs, all children of old link moved to be children of new link
668   → or deleted.
669 /// </summary>
670 public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
671   → TLink newLinkIndex)
672 {
673     var equalityComparer = EqualityComparer<TLink>.Default;
674     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
675     {
676         var constants = links.Constants;
677         var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
678           → constants.Any);
679         long usagesAsSourceCount = (Integer<TLink>)links.Count(usagesAsSourceQuery);
680         var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
681           → oldLinkIndex);
682         long usagesAsTargetCount = (Integer<TLink>)links.Count(usagesAsTargetQuery);
683         var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
684           → usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
685         if (!isStandalonePoint)
686         {
687             var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
688             if (totalUsages > 0)
689             {
690                 var usages = ArrayPool.Allocate<TLink>(totalUsages);
691                 var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
692                   → links.Constants.Continue);
693                 var i = 0L;
694                 if (usagesAsSourceCount > 0)
695                 {
696                     links.Each(usagesFiller.AddFirstAndReturnConstant,
697                       → usagesAsSourceQuery);
698                     for (; i < usagesAsSourceCount; i++)
699                     {
700                         var usage = usages[i];
701                         if (!equalityComparer.Equals(usage, oldLinkIndex))
702                         {
703                             links.Update(usage, newLinkIndex, links.GetTarget(usage));
704                         }
705                     }
706                 }
707                 if (usagesAsTargetCount > 0)
708                 {
709                     links.Each(usagesFiller.AddFirstAndReturnConstant,
710                       → usagesAsTargetQuery);
711                     for (; i < usages.Length; i++)
712                     {
713                         var usage = usages[i];
714                         if (!equalityComparer.Equals(usage, oldLinkIndex))
715                         {
716                             links.Update(usage, links.GetSource(usage), newLinkIndex);
717                         }
718                     }
719                 }
720                 ArrayPool.Free(usages);
721             }
722         }
723     }
724     return newLinkIndex;
725 }
726
727 /// <summary>
728 /// Replace one link with another (replaced link is deleted, children are updated or
729   → deleted).
730 /// </summary>
731 [MethodImpl(MethodImplOptions.AggressiveInlining)]
732 public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
733   → TLink newLinkIndex)
734 {
735     var equalityComparer = EqualityComparer<TLink>.Default;
736     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
737     {
738         links.MergeUsages(oldLinkIndex, newLinkIndex);
739         links.Delete(oldLinkIndex);
740     }
741     return newLinkIndex;
742 }

```

```

732     }
733
734     public static ILinks<TLink>
735     ↪ DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
736     {
737         links = new LinksCascadeUsagesResolver<TLink>(links);
738         links = new NonNullContentsLinkDeletionResolver<TLink>(links);
739         links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
740         return links;
741     }
742 }

```

1.20 ./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>,
6      ↪ LinksConstants<TLink>>, ILinks<TLink>
7      {
8      }
9  }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Incrementers;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Incrementers
7  {
8      public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11         ↪ EqualityComparer<TLink>.Default;
12
13         private readonly TLink _frequencyMarker;
14         private readonly TLink _unaryOne;
15         private readonly IIncrementer<TLink> _unaryNumberIncrementer;
16
17         public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
18         ↪ IIncrementer<TLink> unaryNumberIncrementer)
19         : base(links)
20         {
21             _frequencyMarker = frequencyMarker;
22             _unaryOne = unaryOne;
23             _unaryNumberIncrementer = unaryNumberIncrementer;
24         }
25
26         public TLink Increment(TLink frequency)
27         {
28             if (_equalityComparer.Equals(frequency, default))
29             {
30                 return Links.GetOrCreate(_unaryOne, _frequencyMarker);
31             }
32             var source = Links.GetSource(frequency);
33             var incrementedSource = _unaryNumberIncrementer.Increment(source);
34             return Links.GetOrCreate(incrementedSource, _frequencyMarker);
35         }
36     }
37 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Incrementers;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Incrementers
7  {
8      public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11         ↪ EqualityComparer<TLink>.Default;
12
13         private readonly TLink _unaryOne;
14
15         public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
16         ↪ _unaryOne = unaryOne;
17     }
18 }

```

```

15
16 public TLink Increment(TLink unaryNumber)
17 {
18     if (_equalityComparer.Equals(unaryNumber, _unaryOne))
19     {
20         return Links.GetOrCreate(_unaryOne, _unaryOne);
21     }
22     var source = Links.GetSource(unaryNumber);
23     var target = Links.GetTarget(unaryNumber);
24     if (_equalityComparer.Equals(source, target))
25     {
26         return Links.GetOrCreate(unaryNumber, _unaryOne);
27     }
28     else
29     {
30         return Links.GetOrCreate(source, Increment(target));
31     }
32 }
33 }
34 }

```

1.23 ./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         private const int Length = 3;
27
28         public readonly TLink Index;
29         public readonly TLink Source;
30         public readonly TLink Target;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public Link(params TLink[] values) => SetValues(values, out Index, out Source, out
34             ↪ Target);
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public Link(IList<TLink> values) => SetValues(values, out Index, out Source, out Target);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public Link(object other)
41         {
42             if (other is Link<TLink> otherLink)
43             {
44                 SetValues(ref otherLink, out Index, out Source, out Target);
45             }
46             else if (other is IList<TLink> otherList)
47             {
48                 SetValues(otherList, out Index, out Source, out Target);
49             }
50             else
51             {
52                 throw new NotSupportedException();
53             }
54         }
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         public Link(ref Link<TLink> other) => SetValues(ref other, out Index, out Source, out
58             ↪ Target);
59     }
60 }

```

```

55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 public Link(TLink index, TLink source, TLink target)
57 {
58     Index = index;
59     Source = source;
60     Target = target;
61 }
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 private static void SetValues(ref Link<TLink> other, out TLink index, out TLink source,
65     ↪ out TLink target)
66 {
67     index = other.Index;
68     source = other.Source;
69     target = other.Target;
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 private static void SetValues(IList<TLink> values, out TLink index, out TLink source,
74     ↪ out TLink target)
75 {
76     switch (values.Count)
77     {
78         case 3:
79             index = values[0];
80             source = values[1];
81             target = values[2];
82             break;
83         case 2:
84             index = values[0];
85             source = values[1];
86             target = default;
87             break;
88         case 1:
89             index = values[0];
90             source = default;
91             target = default;
92             break;
93         default:
94             index = default;
95             source = default;
96             target = default;
97             break;
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> &&
110     ↪ Equals((Link<TLink>)other);
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
114     && _equalityComparer.Equals(Source, other.Source)
115     && _equalityComparer.Equals(Target, other.Target);
116
117 [MethodImpl(MethodImplOptions.AggressiveInlining)]
118 public static string ToString(TLink index, TLink source, TLink target) => $"{index}:
119     ↪ {source}->{target}";
120
121 [MethodImpl(MethodImplOptions.AggressiveInlining)]
122 public static string ToString(TLink source, TLink target) => $"{source}->{target}";
123
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 public static implicit operator TLink[] (Link<TLink> link) => link.ToArray();
126
127 [MethodImpl(MethodImplOptions.AggressiveInlining)]
128 public static implicit operator Link<TLink>(TLink[] linkArray) => new
    ↪ Link<TLink>(linkArray);

```

```

129 public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
    ↳ ToString(Source, Target) : ToString(Index, Source, Target);
130
131 #region IList
132
133 public int Count => Length;
134
135 public bool IsReadOnly => true;
136
137 public TLink this[int index]
138 {
139     [MethodImpl(MethodImplOptions.AggressiveInlining)]
140     get
141     {
142         Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
143             ↳ nameof(index));
144         if (index == _constants.IndexPart)
145         {
146             return Index;
147         }
148         if (index == _constants.SourcePart)
149         {
150             return Source;
151         }
152         if (index == _constants.TargetPart)
153         {
154             return Target;
155         }
156         throw new NotSupportedException(); // Impossible path due to
157             ↳ Ensure.ArgumentInRange
158     }
159     [MethodImpl(MethodImplOptions.AggressiveInlining)]
160     set => throw new NotSupportedException();
161 }
162
163 [MethodImpl(MethodImplOptions.AggressiveInlining)]
164 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
165
166 [MethodImpl(MethodImplOptions.AggressiveInlining)]
167 public IEnumerator<TLink> GetEnumerator()
168 {
169     yield return Index;
170     yield return Source;
171     yield return Target;
172 }
173
174 [MethodImpl(MethodImplOptions.AggressiveInlining)]
175 public void Add(TLink item) => throw new NotSupportedException();
176
177 [MethodImpl(MethodImplOptions.AggressiveInlining)]
178 public void Clear() => throw new NotSupportedException();
179
180 [MethodImpl(MethodImplOptions.AggressiveInlining)]
181 public bool Contains(TLink item) => IndexOf(item) >= 0;
182
183 [MethodImpl(MethodImplOptions.AggressiveInlining)]
184 public void CopyTo(TLink[] array, int arrayIndex)
185 {
186     Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
187     Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
188         ↳ nameof(arrayIndex));
189     if (arrayIndex + Length > array.Length)
190     {
191         throw new InvalidOperationException();
192     }
193     array[arrayIndex++] = Index;
194     array[arrayIndex++] = Source;
195     array[arrayIndex] = Target;
196 }
197
198 [MethodImpl(MethodImplOptions.AggressiveInlining)]
199 public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
200
201 [MethodImpl(MethodImplOptions.AggressiveInlining)]
202 public int IndexOf(TLink item)
203 {
204     if (_equalityComparer.Equals(Index, item))
205     {
206         return _constants.IndexPart;
207     }
208     if (_equalityComparer.Equals(Source, item))
209     {
210         return _constants.SourcePart;
211     }
212     if (_equalityComparer.Equals(Target, item))
213     {
214         return _constants.TargetPart;
215     }
216     return -1;
217 }

```

```

204     }
205     if (_equalityComparer.Equals(Source, item))
206     {
207         return _constants.SourcePart;
208     }
209     if (_equalityComparer.Equals(Target, item))
210     {
211         return _constants.TargetPart;
212     }
213     return -1;
214 }
215
216 [MethodImpl(MethodImplOptions.AggressiveInlining)]
217 public void Insert(int index, TLink item) => throw new NotSupportedException();
218
219 [MethodImpl(MethodImplOptions.AggressiveInlining)]
220 public void RemoveAt(int index) => throw new NotSupportedException();
221
222 #endregion
223 }
224 }

```

1.24 ./Platform.Data.Doublets/LinkExtensions.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets
4  {
5      public static class LinkExtensions
6      {
7          public static bool IsFullPoint<TLink>(this Link<TLink> link) =>
8              ↪ Point<TLink>.IsFullPoint(link);
9          public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
10             ↪ Point<TLink>.IsPartialPoint(link);
11     }
12 }

```

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

```

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

```

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

```

1  using System.Collections.Generic;
2  using Platform.Reflection;
3  using Platform.Converters;
4  using Platform.Numbers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Numbers.Unary
9  {
10     public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
11         ↪ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↪ EqualityComparer<TLink>.Default;
15
16         private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
17
18         public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
19             ↪ powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
20             ↪ powerOf2ToUnaryNumberConverter;
21
22         public TLink Convert(TLink number)
23         {
24             var nullConstant = Links.Constants.Null;
25             var one = Integer<TLink>.One;
26             var target = nullConstant;
27             for (int i = 0; !_equalityComparer.Equals(number, default) && i <
28                 ↪ NumericType<TLink>.BitsSize; i++)
29             {
30                 if (_equalityComparer.Equals(Bit.And(number, one), one))

```

```

26         {
27             target = _equalityComparer.Equals(target, nullConstant)
28                 ? _powerOf2ToUnaryNumberConverter.Convert(i)
29                 : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
30         }
31         number = Bit.ShiftRight(number, 1);
32     }
33     return target;
34 }
35 }
36 }

```

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

```

1  using System;
2  using System.Collections.Generic;
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.Numbers.Unary
9  {
10     public class LinkToItsFrequencyNumberConverter<TLink> : LinksOperatorBase<TLink>,
11         ↪ IConverter<Doublet<TLink>, TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↪ EqualityComparer<TLink>.Default;
15
16         private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
17         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
18
19         public LinkToItsFrequencyNumberConverter(
20             ILinks<TLink> links,
21             IProperty<TLink, TLink> frequencyPropertyOperator,
22             IConverter<TLink> unaryNumberToAddressConverter)
23             : base(links)
24         {
25             _frequencyPropertyOperator = frequencyPropertyOperator;
26             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
27         }
28
29         public TLink Convert(Doublet<TLink> doublet)
30         {
31             var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
32             if (_equalityComparer.Equals(link, default))
33             {
34                 throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
35             }
36             var frequency = _frequencyPropertyOperator.Get(link);
37             if (_equalityComparer.Equals(frequency, default))
38             {
39                 return default;
40             }
41             var frequencyNumber = Links.GetSource(frequency);
42             return _unaryNumberToAddressConverter.Convert(frequencyNumber);
43         }
44     }
45 }

```

1.28 ./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
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 PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
11         ↪ IConverter<int, TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↪ EqualityComparer<TLink>.Default;
15
16         private readonly TLink[] _unaryNumberPowersOf2;
17
18         public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
19         {
20             _unaryNumberPowersOf2 = new TLink[64];
21         }
22     }
23 }

```

```

19     _unaryNumberPowersOf2[0] = one;
20 }
21
22 public TLink Convert(int power)
23 {
24     Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
    ↪ - 1), nameof(power));
25     if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
26     {
27         return _unaryNumberPowersOf2[power];
28     }
29     var previousPowerOf2 = Convert(power - 1);
30     var powerOf2 = Links.GetOrCreate(previousPowerOf2, previousPowerOf2);
31     _unaryNumberPowersOf2[power] = powerOf2;
32     return powerOf2;
33 }
34 }
35 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Converters;
4 using Platform.Numbers;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Numbers.Unary
9 {
10     public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↪ EqualityComparer<TLink>.Default;
13
14         private Dictionary<TLink, TLink> _unaryToUInt64;
15         private readonly TLink _unaryOne;
16
17         public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
18             : base(links)
19         {
20             _unaryOne = unaryOne;
21             InitUnaryToUInt64();
22         }
23
24         private void InitUnaryToUInt64()
25         {
26             var one = Integer<TLink>.One;
27             _unaryToUInt64 = new Dictionary<TLink, TLink>
28             {
29                 { _unaryOne, one }
30             };
31             var unary = _unaryOne;
32             var number = one;
33             for (var i = 1; i < 64; i++)
34             {
35                 unary = Links.GetOrCreate(unary, unary);
36                 number = Double(number);
37                 _unaryToUInt64.Add(unary, number);
38             }
39         }
40
41         public TLink Convert(TLink unaryNumber)
42         {
43             if (_equalityComparer.Equals(unaryNumber, default))
44             {
45                 return default;
46             }
47             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
48             {
49                 return Integer<TLink>.One;
50             }
51             var source = Links.GetSource(unaryNumber);
52             var target = Links.GetTarget(unaryNumber);
53             if (_equalityComparer.Equals(source, target))
54             {
55                 return _unaryToUInt64[unaryNumber];
56             }
57             else
58             {

```



```

59     var result = _unaryToUInt64[source];
60     TLink lastValue;
61     while (!_unaryToUInt64.TryGetValue(target, out lastValue))
62     {
63         source = Links.GetSource(target);
64         result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
65         target = Links.GetTarget(target);
66     }
67     result = Arithmetic<TLink>.Add(result, lastValue);
68     return result;
69 }
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 private static TLink Double(TLink number) => (Integer<TLink>)((Integer<TLink>)number *
    ↪ 2UL);
74 }
75 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Reflection;
4  using Platform.Converters;
5  using Platform.Numbers;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↪ EqualityComparer<TLink>.Default;
14
15         private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
16
17         public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
    ↪ TLink> powerOf2ToUnaryNumberConverter)
    : base(links)
18         {
19             _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
20             for (int i = 0; i < NumericType<TLink>.BitsSize; i++)
21             {
22                 _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
23             }
24         }
25
26         public TLink Convert(TLink sourceNumber)
27         {
28             var nullConstant = Links.Constants.Null;
29             var source = sourceNumber;
30             var target = nullConstant;
31             if (!_equalityComparer.Equals(source, nullConstant))
32             {
33                 while (true)
34                 {
35                     if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
36                     {
37                         SetBit(ref target, powerOf2Index);
38                         break;
39                     }
40                     else
41                     {
42                         powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
43                         SetBit(ref target, powerOf2Index);
44                         source = Links.GetTarget(source);
45                     }
46                 }
47             }
48             return target;
49         }
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         private static void SetBit(ref TLink target, int powerOf2Index) => target =
    ↪ Bit.Or(target, Bit.ShiftLeft(Integer<TLink>.One, powerOf2Index));
53
54     }
55 }

```

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

```

1  using System.Linq;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.PropertyOperators
8  {
9      public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>, IProperties<TLink, TLink,
10         ↪ TLink>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↪ EqualityComparer<TLink>.Default;
14
15          public PropertiesOperator(ILinks<TLink> links) : base(links) { }
16
17          public TLink GetValue(TLink @object, TLink property)
18          {
19              var objectProperty = Links.SearchOrDefault(@object, property);
20              if (_equalityComparer.Equals(objectProperty, default))
21              {
22                  return default;
23              }
24              var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
25              if (valueLink == null)
26              {
27                  return default;
28              }
29              return Links.GetTarget(valueLink[Links.Constants.IndexPart]);
30          }
31
32          public void SetValue(TLink @object, TLink property, TLink value)
33          {
34              var objectProperty = Links.GetOrCreate(@object, property);
35              Links.DeleteMany(Links.AllIndices(Links.Constants.Any, objectProperty));
36              Links.GetOrCreate(objectProperty, value);
37          }
38      }
39  }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.PropertyOperators
7  {
8      public class PropertyOperator<TLink> : LinksOperatorBase<TLink>, IProperty<TLink, TLink>
9      {
10          private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↪ EqualityComparer<TLink>.Default;
12
13          private readonly TLink _propertyMarker;
14          private readonly TLink _propertyValueMarker;
15
16          public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
17             ↪ propertyValueMarker) : base(links)
18          {
19              _propertyMarker = propertyMarker;
20              _propertyValueMarker = propertyValueMarker;
21          }
22
23          public TLink Get(TLink link)
24          {
25              var property = Links.SearchOrDefault(link, _propertyMarker);
26              var container = GetContainer(property);
27              var value = GetValue(container);
28              return value;
29          }
30
31          private TLink GetContainer(TLink property)
32          {
33              var valueContainer = default(TLink);
34              if (_equalityComparer.Equals(property, default))
35              {
36                  return valueContainer;
37              }
38              var constants = Links.Constants;

```

```

37     var continueConstant = constants.Continue;
38     var breakConstant = constants.Break;
39     var anyConstant = constants.Any;
40     var query = new Link<TLink>(anyConstant, property, anyConstant);
41     Links.Each(candidate =>
42     {
43         var candidateTarget = Links.GetTarget(candidate);
44         var valueTarget = Links.GetTarget(candidateTarget);
45         if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
46         {
47             valueContainer = Links.GetIndex(candidate);
48             return breakConstant;
49         }
50         return continueConstant;
51     }, query);
52     return valueContainer;
53 }
54
55 private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
56     ? default : Links.GetTarget(container);
57
58 public void Set(TLink link, TLink value)
59 {
60     var property = Links.GetOrCreate(link, _propertyMarker);
61     var container = GetContainer(property);
62     if (_equalityComparer.Equals(container, default))
63     {
64         Links.GetOrCreate(property, value);
65     }
66     else
67     {
68         Links.Update(container, property, value);
69     }
70 }
71 }

```

1.33 ./Platform.Data.Doublets/ResizableDirectMemory/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.Numbers;
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.ResizableDirectMemory.Generic
12 {
13     public unsafe abstract class LinksAvlBalancedTreeMethodsBase<TLink> :
14         ↳ SizedAndThreadedAVLBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
15     {
16         protected readonly TLink Break;
17         protected readonly TLink Continue;
18         protected readonly byte* Links;
19         protected readonly byte* Header;
20
21         public LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
22             ↳ byte* header)
23         {
24             Links = links;
25             Header = header;
26             Break = constants.Break;
27             Continue = constants.Continue;
28
29             [MethodImpl(MethodImplOptions.AggressiveInlining)]
30             protected abstract TLink GetTreeRoot();
31
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             protected abstract TLink GetBasePartValue(TLink link);
34
35             [MethodImpl(MethodImplOptions.AggressiveInlining)]
36             protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
37                 ↳ rootSource, TLink rootTarget);
38
39             [MethodImpl(MethodImplOptions.AggressiveInlining)]
40             protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
41                 ↳ rootSource, TLink rootTarget);

```

```

39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
41     ↳ AsRef<LinksHeader<TLink>>(Header);
42
43 [MethodImpl(MethodImplOptions.AggressiveInlining)]
44 protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
45     ↳ AsRef<RawLink<TLink>>(Links + RawLink<TLink>.SizeInBytes * (Integer<TLink>)link);
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
49 {
50     ref var link = ref GetLinkReference(linkIndex);
51     return new Link<TLink>(linkIndex, link.Source, link.Target);
52 }
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
56 {
57     ref var firstLink = ref GetLinkReference(first);
58     ref var secondLink = ref GetLinkReference(second);
59     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
60     ↳ secondLink.Source, secondLink.Target);
61 }
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
65 {
66     ref var firstLink = ref GetLinkReference(first);
67     ref var secondLink = ref GetLinkReference(second);
68     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
69     ↳ secondLink.Source, secondLink.Target);
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 protected virtual TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5,
74     ↳ -5);
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 protected virtual void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
78     ↳ Bit<TLink>.PartialWrite(storedValue, size, 5, -5);
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 protected virtual bool GetLeftIsChildValue(TLink value)
82 {
83     unchecked
84     {
85         //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 4, 1);
86         return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
87     }
88 }
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 protected virtual void SetLeftIsChildValue(ref TLink storedValue, bool value)
92 {
93     unchecked
94     {
95         var previousValue = storedValue;
96         var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 4,
97         ↳ 1);
98         storedValue = modified;
99     }
100 }
101
102 [MethodImpl(MethodImplOptions.AggressiveInlining)]
103 protected virtual bool GetRightIsChildValue(TLink value)
104 {
105     unchecked
106     {
107         //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 3, 1);
108         return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
109     }
110 }
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 protected virtual void SetRightIsChildValue(ref TLink storedValue, bool value)
114 {
115     unchecked

```

```

110     {
111         var previousValue = storedValue;
112         var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 3,
            ↪ 1);
113         storedValue = modified;
114     }
115 }
116
117 [MethodImpl(MethodImplOptions.AggressiveInlining)]
118 protected bool IsChild(TLink parent, TLink possibleChild)
119 {
120     var parentSize = GetSize(parent);
121     var childSize = GetSizeOrZero(possibleChild);
122     return GreaterThanZero(childSize) && LessOrEqualThan(childSize, parentSize);
123 }
124
125 [MethodImpl(MethodImplOptions.AggressiveInlining)]
126 protected virtual sbyte GetBalanceValue(TLink storedValue)
127 {
128     unchecked
129     {
130         var value = (int)(Integer<TLink>)Bit<TLink>.PartialRead(storedValue, 0, 3);
131         value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
            ↪ end of sbyte
132         return (sbyte)value;
133     }
134 }
135
136 [MethodImpl(MethodImplOptions.AggressiveInlining)]
137 protected virtual void SetBalanceValue(ref TLink storedValue, sbyte value)
138 {
139     unchecked
140     {
141         var packagedValue = (TLink)(Integer<TLink>)((byte)value >> 5 & 4 | value & 3);
142         var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
143         storedValue = modified;
144     }
145 }
146
147 public TLink this[TLink index]
148 {
149     get
150     {
151         var root = GetTreeRoot();
152         if (GreaterOrEqualThan(index, GetSize(root)))
153         {
154             return Zero;
155         }
156         while (!EqualToZero(root))
157         {
158             var left = GetLeftOrDefault(root);
159             var leftSize = GetSizeOrZero(left);
160             if (LessThan(index, leftSize))
161             {
162                 root = left;
163                 continue;
164             }
165             if (AreEqual(index, leftSize))
166             {
167                 return root;
168             }
169             root = GetRightOrDefault(root);
170             index = Subtract(index, Increment(leftSize));
171         }
172         return Zero; // TODO: Impossible situation exception (only if tree structure
            ↪ broken)
173     }
174 }
175
176 /// <summary>
177 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
178 ↪ (концом).
179 /// </summary>
180 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
181 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
182 /// <returns>Индекс искомой связи.</returns>
183 public TLink Search(TLink source, TLink target)
184 {
185     var root = GetTreeRoot();

```

```

185 while (!EqualToZero(root))
186 {
187     ref var rootLink = ref GetLinkReference(root);
188     var rootSource = rootLink.Source;
189     var rootTarget = rootLink.Target;
190     if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
191         ↪ node.Key < root.Key
192     {
193         root = GetLeftOrDefault(root);
194     }
195     else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
196         ↪ node.Key > root.Key
197     {
198         root = GetRightOrDefault(root);
199     }
200     else // node.Key == root.Key
201     {
202         return root;
203     }
204 }
205 return Zero;
206 }
207 // TODO: Return indices range instead of references count
208 public TLink CountUsages(TLink link)
209 {
210     var root = GetTreeRoot();
211     var total = GetSize(root);
212     var totalRightIgnore = Zero;
213     while (!EqualToZero(root))
214     {
215         var @base = GetBasePartValue(root);
216         if (LessOrEqualThan(@base, link))
217         {
218             root = GetRightOrDefault(root);
219         }
220         else
221         {
222             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
223             root = GetLeftOrDefault(root);
224         }
225     }
226     root = GetTreeRoot();
227     var totalLeftIgnore = Zero;
228     while (!EqualToZero(root))
229     {
230         var @base = GetBasePartValue(root);
231         if (GreaterOrEqualThan(@base, link))
232         {
233             root = GetLeftOrDefault(root);
234         }
235         else
236         {
237             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
238             root = GetRightOrDefault(root);
239         }
240     }
241     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
242 }
243
244 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
245 {
246     var root = GetTreeRoot();
247     if (EqualToZero(root))
248     {
249         return Continue;
250     }
251     TLink first = Zero, current = root;
252     while (!EqualToZero(current))
253     {
254         var @base = GetBasePartValue(current);
255         if (GreaterOrEqualThan(@base, link))
256         {
257             if (AreEqual(@base, link))
258             {
259                 first = current;
260             }
261             current = GetLeftOrDefault(current);

```

```

262     }
263     else
264     {
265         current = GetRightOrDefault(current);
266     }
267 }
268 if (!EqualToZero(first))
269 {
270     current = first;
271     while (true)
272     {
273         if (AreEqual(handler(GetLinkValues(current)), Break))
274         {
275             return Break;
276         }
277         current = GetNext(current);
278         if (EqualToZero(current) || !AreEqual(GetBasePartValue(current), link))
279         {
280             break;
281         }
282     }
283 }
284 return Continue;
285 }
286
287 protected override void PrintNodeValue(TLink node, StringBuilder sb)
288 {
289     ref var link = ref GetLinkReference(node);
290     sb.Append(' ');
291     sb.Append(link.Source);
292     sb.Append('-');
293     sb.Append('>');
294     sb.Append(link.Target);
295 }
296 }
297 }

```

1.34 ./Platform.Data.Doublets/ResizableDirectMemory/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.Numbers;
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.ResizableDirectMemory.Generic
12 {
13     public unsafe abstract class LinksSizeBalancedTreeMethodsBase<TLink> :
14         ↳ SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
15     {
16         protected readonly TLink Break;
17         protected readonly TLink Continue;
18         protected readonly byte* Links;
19         protected readonly byte* Header;
20
21         public LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
22             ↳ byte* header)
23         {
24             Links = links;
25             Header = header;
26             Break = constants.Break;
27             Continue = constants.Continue;
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected abstract TLink GetTreeRoot();
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected abstract TLink GetBasePartValue(TLink link);
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
38             ↳ rootSource, TLink rootTarget);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
42             ↳ rootSource, TLink rootTarget);
43     }
44 }

```

```

39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
41     ↳ AsRef<LinksHeader<TLink>>(Header);
42
43 [MethodImpl(MethodImplOptions.AggressiveInlining)]
44 protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
45     ↳ AsRef<RawLink<TLink>>(Links + RawLink<TLink>.SizeInBytes * (Integer<TLink>)link);
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
49 {
50     ref var link = ref GetLinkReference(linkIndex);
51     return new Link<TLink>(linkIndex, link.Source, link.Target);
52 }
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
56 {
57     ref var firstLink = ref GetLinkReference(first);
58     ref var secondLink = ref GetLinkReference(second);
59     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
60     ↳ secondLink.Source, secondLink.Target);
61 }
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
65 {
66     ref var firstLink = ref GetLinkReference(first);
67     ref var secondLink = ref GetLinkReference(second);
68     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
69     ↳ secondLink.Source, secondLink.Target);
70 }
71
72 public TLink this[TLink index]
73 {
74     get
75     {
76         var root = GetTreeRoot();
77         if (GreaterOrEqualThan(index, GetSize(root)))
78         {
79             return Zero;
80         }
81         while (!EqualToZero(root))
82         {
83             var left = GetLeftOrDefault(root);
84             var leftSize = GetSizeOrZero(left);
85             if (LessThan(index, leftSize))
86             {
87                 root = left;
88                 continue;
89             }
90             if (AreEqual(index, leftSize))
91             {
92                 return root;
93             }
94             root = GetRightOrDefault(root);
95             index = Subtract(index, Increment(leftSize));
96         }
97         return Zero; // TODO: Impossible situation exception (only if tree structure
98     ↳ broken)
99     }
100 }
101
102 /// <summary>
103 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
104 ↳ (концом).
105 /// </summary>
106 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
107 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
108 /// <returns>Индекс искомой связи.</returns>
109 public TLink Search(TLink source, TLink target)
110 {
111     var root = GetTreeRoot();
112     while (!EqualToZero(root))
113     {
114         ref var rootLink = ref GetLinkReference(root);
115         var rootSource = rootLink.Source;

```



```

111     var rootTarget = rootLink.Target;
112     if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
113         ↪ node.Key < root.Key
114     {
115         root = GetLeftOrDefault(root);
116     }
117     else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
118         ↪ node.Key > root.Key
119     {
120         root = GetRightOrDefault(root);
121     }
122     else // node.Key == root.Key
123     {
124         return root;
125     }
126 }
127 return Zero;
128 }
129 // TODO: Return indices range instead of references count
130 public TLink CountUsages(TLink link)
131 {
132     var root = GetTreeRoot();
133     var total = GetSize(root);
134     var totalRightIgnore = Zero;
135     while (!EqualToZero(root))
136     {
137         var @base = GetBasePartValue(root);
138         if (LessOrEqualThan(@base, link))
139         {
140             root = GetRightOrDefault(root);
141         }
142         else
143         {
144             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
145             root = GetLeftOrDefault(root);
146         }
147     }
148     root = GetTreeRoot();
149     var totalLeftIgnore = Zero;
150     while (!EqualToZero(root))
151     {
152         var @base = GetBasePartValue(root);
153         if (GreaterOrEqualThan(@base, link))
154         {
155             root = GetLeftOrDefault(root);
156         }
157         else
158         {
159             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
160             root = GetRightOrDefault(root);
161         }
162     }
163     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
164 }
165
166 [MethodImpl(MethodImplOptions.AggressiveInlining)]
167 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
168     ↪ EachUsageCore(@base, GetTreeRoot(), handler);
169
170 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
171 ↪ low-level MSIL stack.
172 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
173 {
174     var @continue = Continue;
175     if (EqualToZero(link))
176     {
177         return @continue;
178     }
179     var linkBasePart = GetBasePartValue(link);
180     var @break = Break;
181     if (GreaterThan(linkBasePart, @base))
182     {
183         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
184         {
185             return @break;
186         }
187     }
188 }

```

```

186     else if (LessThan(linkBasePart, @base))
187     {
188         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
189         {
190             return @break;
191         }
192     }
193     else //if (linkBasePart == @base)
194     {
195         if (AreEqual(handler(GetLinkValues(link)), @break))
196         {
197             return @break;
198         }
199         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
200         {
201             return @break;
202         }
203         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
204         {
205             return @break;
206         }
207     }
208     return @continue;
209 }
210
211 protected override void PrintNodeValue(TLink node, StringBuilder sb)
212 {
213     ref var link = ref GetLinkReference(node);
214     sb.Append(' ');
215     sb.Append(link.Source);
216     sb.Append('-');
217     sb.Append('>');
218     sb.Append(link.Target);
219 }
220 }
221 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
6 {
7     public unsafe class LinksSourcesAvlBalancedTreeMethods<TLink> :
8     ↪ LinksAvlBalancedTreeMethodsBase<TLink>
9     {
10         public LinksSourcesAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
11         ↪ byte* header) : base(constants, links, header) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected unsafe override ref TLink GetLeftReference(TLink node) => ref
15         ↪ GetLinkReference(node).LeftAsSource;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected unsafe override ref TLink GetRightReference(TLink node) => ref
19         ↪ GetLinkReference(node).RightAsSource;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override void SetLeft(TLink node, TLink left) =>
29         ↪ GetLinkReference(node).LeftAsSource = left;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetRight(TLink node, TLink right) =>
33         ↪ GetLinkReference(node).RightAsSource = right;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override TLink GetSize(TLink node) =>
37         ↪ GetSizeValue(GetLinkReference(node).SizeAsSource);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
41         ↪ GetLinkReference(node).SizeAsSource, size);
42     }
43 }

```

```

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

```

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

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
6  {
7      public unsafe class LinksSourcesSizeBalancedTreeMethods<TLink> :
8      ↪ LinksSizeBalancedTreeMethodsBase<TLink>
9      {
10         public LinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
11         ↪ byte* header) : base(constants, links, header) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected unsafe override ref TLink GetLeftReference(TLink node) => ref
15         ↪ GetLinkReference(node).LeftAsSource;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected unsafe override ref TLink GetRightReference(TLink node) => ref
19         ↪ GetLinkReference(node).RightAsSource;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
26     }
27 }

```

```

22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected override void SetLeft(TLink node, TLink left) =>
24         ↳ GetLinkReference(node).LeftAsSource = left;
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected override void SetRight(TLink node, TLink right) =>
28         ↳ 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) =>
35         ↳ GetLinkReference(node).SizeAsSource = size;
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsSource;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
45         ↳ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
46         ↳ AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget);
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
50         ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
51         ↳ AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget);
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override void ClearNode(TLink node)
55     {
56         ref var link = ref GetLinkReference(node);
57         link.LeftAsSource = Zero;
58         link.RightAsSource = Zero;
59         link.SizeAsSource = Zero;
60     }
61 }

```

1.37 ./Platform.Data.Doublets/ResizableDirectMemory/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.ResizableDirectMemory.Generic
6  {
7      public unsafe class LinksTargetsAvlBalancedTreeMethods<TLink> :
8          ↳ LinksAvlBalancedTreeMethodsBase<TLink>
9      {
10         public LinksTargetsAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
11             ↳ byte* header) : base(constants, links, header) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected unsafe override ref TLink GetLeftReference(TLink node) => ref
15             ↳ GetLinkReference(node).LeftAsTarget;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected unsafe override ref TLink GetRightReference(TLink node) => ref
19             ↳ GetLinkReference(node).RightAsTarget;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override void SetLeft(TLink node, TLink left) =>
29             ↳ GetLinkReference(node).LeftAsTarget = left;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetRight(TLink node, TLink right) =>
33             ↳ GetLinkReference(node).RightAsTarget = right;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

30     protected override TLink GetSize(TLink node) =>
31         ↪ GetSizeValue(GetLinkReference(node).SizeAsTarget);
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
35         ↪ GetLinkReference(node).SizeAsTarget, size);
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override bool GetLeftIsChild(TLink node) =>
39         ↪ GetLeftIsChildValue(GetLinkReference(node).SizeAsTarget);
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override void SetLeftIsChild(TLink node, bool value) =>
43         ↪ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override bool GetRightIsChild(TLink node) =>
47         ↪ GetRightIsChildValue(GetLinkReference(node).SizeAsTarget);
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override void SetRightIsChild(TLink node, bool value) =>
51         ↪ SetRightIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override sbyte GetBalance(TLink node) =>
55         ↪ GetBalanceValue(GetLinkReference(node).SizeAsTarget);
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
59         ↪ GetLinkReference(node).SizeAsTarget, value);
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsTarget;
63
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
66
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
69         ↪ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
70         ↪ AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource);
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
74         ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
75         ↪ AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource);
76
77     [MethodImpl(MethodImplOptions.AggressiveInlining)]
78     protected override void ClearNode(TLink node)
79     {
80         ref var link = ref GetLinkReference(node);
81         link.LeftAsTarget = Zero;
82         link.RightAsTarget = Zero;
83         link.SizeAsTarget = Zero;
84     }
85 }

```

1.38 ./Platform.Data.Doublets/ResizableDirectMemory/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.ResizableDirectMemory.Generic
6     {
7         public unsafe class LinksTargetsSizeBalancedTreeMethods<TLink> :
8             ↪ LinksSizeBalancedTreeMethodsBase<TLink>
9         {
10             public LinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
11                 ↪ byte* header) : base(constants, links, header) { }
12
13             [MethodImpl(MethodImplOptions.AggressiveInlining)]
14             protected unsafe override ref TLink GetLeftReference(TLink node) => ref
15                 ↪ GetLinkReference(node).LeftAsTarget;
16
17             [MethodImpl(MethodImplOptions.AggressiveInlining)]
18             protected unsafe override ref TLink GetRightReference(TLink node) => ref
19                 ↪ GetLinkReference(node).RightAsTarget;
20
21             [MethodImpl(MethodImplOptions.AggressiveInlining)]
22             protected unsafe override void SetLeftReference(TLink node, ref TLink value) =>
23                 ↪ SetLinkReference(ref GetLinkReference(node).LeftAsTarget, value);
24
25             [MethodImpl(MethodImplOptions.AggressiveInlining)]
26             protected unsafe override void SetRightReference(TLink node, ref TLink value) =>
27                 ↪ SetLinkReference(ref GetLinkReference(node).RightAsTarget, value);
28
29             [MethodImpl(MethodImplOptions.AggressiveInlining)]
30             protected unsafe override void ClearNode(TLink node)
31             {
32                 ref var link = ref GetLinkReference(node);
33                 link.LeftAsTarget = Zero;
34                 link.RightAsTarget = Zero;
35                 link.SizeAsTarget = Zero;
36             }
37         }
38     }

```

```

16 [MethodImpl(MethodImplOptions.AggressiveInlining)]
17 protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
18
19 [MethodImpl(MethodImplOptions.AggressiveInlining)]
20 protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
21
22 [MethodImpl(MethodImplOptions.AggressiveInlining)]
23 protected override void SetLeft(TLink node, TLink left) =>
24     ↳ GetLinkReference(node).LeftAsTarget = left;
25
26 [MethodImpl(MethodImplOptions.AggressiveInlining)]
27 protected override void SetRight(TLink node, TLink right) =>
28     ↳ GetLinkReference(node).RightAsTarget = right;
29
30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsTarget;
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 protected override void SetSize(TLink node, TLink size) =>
35     ↳ GetLinkReference(node).SizeAsTarget = size;
36
37 [MethodImpl(MethodImplOptions.AggressiveInlining)]
38 protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsTarget;
39
40 [MethodImpl(MethodImplOptions.AggressiveInlining)]
41 protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
42
43 [MethodImpl(MethodImplOptions.AggressiveInlining)]
44 protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
45     ↳ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
46     ↳ AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource);
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
50     ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
51     ↳ AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource);
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected override void ClearNode(TLink node)
55 {
56     ref var link = ref GetLinkReference(node);
57     link.LeftAsTarget = Zero;
58     link.RightAsTarget = Zero;
59     link.SizeAsTarget = Zero;
60 }
61 }
62 }

```

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

```

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

```

```

27 [MethodImpl(MethodImplOptions.AggressiveInlining)]
28 public ResizableDirectMemoryLinks(string address, long memoryReservationStep) : this(new
    ↳ FileMappedResizableDirectMemory(address, memoryReservationStep),
    ↳ memoryReservationStep) { }
29
30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 public ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
    ↳ DefaultLinksSizeStep) { }
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
    ↳ memoryReservationStep) : this(memory, memoryReservationStep,
    ↳ Default<LinksConstants<TLink>>.Instance, true) { }
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
    ↳ memoryReservationStep, LinksConstants<TLink> constants, bool useAvlBasedIndex) :
    ↳ base(memory, memoryReservationStep, constants)
38 {
39     if (useAvlBasedIndex)
40     {
41         _createSourceTreeMethods = () => new
            ↳ LinksSourcesAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
42         _createTargetTreeMethods = () => new
            ↳ LinksTargetsAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
43     }
44     else
45     {
46         _createSourceTreeMethods = () => new
            ↳ LinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
47         _createTargetTreeMethods = () => new
            ↳ LinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
48     }
49     Init(memory, memoryReservationStep);
50 }
51
52 [MethodImpl(MethodImplOptions.AggressiveInlining)]
53 protected override void SetPointers(IResizableDirectMemory memory)
54 {
55     _links = (byte*)memory.Pointer;
56     _header = _links;
57     SourcesTreeMethods = _createSourceTreeMethods();
58     TargetsTreeMethods = _createTargetTreeMethods();
59     UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_links, _header);
60 }
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override void ResetPointers()
64 {
65     base.ResetPointers();
66     _links = null;
67     _header = null;
68 }
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 protected override ref LinksHeader<TLink> GetHeaderReference() => ref
    ↳ AsRef<LinksHeader<TLink>>(_header);
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 protected override ref RawLink<TLink> GetLinkReference(TLink linkIndex) => ref
    ↳ AsRef<RawLink<TLink>>(_links + LinkSizeInBytes * (Integer<TLink>)linkIndex);
75 }
76 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Disposables;
5 using Platform.Singletons;
6 using Platform.Numbers;
7 using Platform.Memory;
8 using Platform.Data.Exceptions;
9
10
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
14 {

```

```

15 public abstract class ResizableDirectMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16 {
17     protected static readonly EqualityComparer<TLink> EqualityComparer =
18         ↳ EqualityComparer<TLink>.Default;
19     protected static readonly Comparer<TLink> Comparer = Comparer<TLink>.Default;
20
21     /// <summary>Возвращает размер одной связи в байтах.</summary>
22     /// <remarks>
23     /// Используется только во вне класса, не рекомендуется использовать внутри.
24     /// Так как во вне не обязательно будет доступен unsafe C#.
25     /// </remarks>
26     public static readonly long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;
27
28     public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
29
30     public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
31
32     protected readonly IResizableDirectMemory _memory;
33     protected readonly long _memoryReservationStep;
34
35     protected ILinksTreeMethods<TLink> TargetsTreeMethods;
36     protected ILinksTreeMethods<TLink> SourcesTreeMethods;
37     // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
38     // ↳ нужно использовать не список а дерево, так как так можно быстрее проверить на
39     // ↳ наличие связи внутри
40     protected ILinksListMethods<TLink> UnusedLinksListMethods;
41
42     /// <summary>
43     /// Возвращает общее число связей находящихся в хранилище.
44     /// </summary>
45     protected virtual TLink Total
46     {
47         get
48         {
49             ref var header = ref GetHeaderReference();
50             return Subtract(header.AllocatedLinks, header.FreeLinks);
51         }
52     }
53
54     public virtual LinksConstants<TLink> Constants { get; }
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     public ResizableDirectMemoryLinksBase(IResizableDirectMemory memory, long
58         ↳ memoryReservationStep, LinksConstants<TLink> constants)
59     {
60         _memory = memory;
61         _memoryReservationStep = memoryReservationStep;
62         Constants = constants;
63     }
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     public ResizableDirectMemoryLinksBase(IResizableDirectMemory memory, long
67         ↳ memoryReservationStep) : this(memory, memoryReservationStep,
68         ↳ Default<LinksConstants<TLink>>.Instance) { }
69
70     protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
71     {
72         if (memory.ReservedCapacity < memoryReservationStep)
73         {
74             memory.ReservedCapacity = memoryReservationStep;
75         }
76         SetPointers(_memory);
77         ref var header = ref GetHeaderReference();
78         // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
79         _memory.UsedCapacity = ConvertToUInt64(header.AllocatedLinks) * LinkSizeInBytes +
80             ↳ LinkHeaderSizeInBytes;
81         // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
82         header.ReservedLinks = ConvertToAddress((_memory.ReservedCapacity -
83             ↳ LinkHeaderSizeInBytes) / LinkSizeInBytes);
84     }
85
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     public virtual TLink Count(IList<TLink> restrictions)
88     {
89         // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
90         if (restrictions.Count == 0)
91         {
92             return Total;
93         }
94         var constants = Constants;

```



```

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

```

```

163         {
164             if (AreEqual(storedLinkValue.Source, source) &&
165                 ↪ AreEqual(storedLinkValue.Target, target))
166             {
167                 return GetOne();
168             }
169             return GetZero();
170         }
171         var value = default(TLink);
172         if (AreEqual(source, any))
173         {
174             value = target;
175         }
176         if (AreEqual(target, any))
177         {
178             value = source;
179         }
180         if (AreEqual(storedLinkValue.Source, value) ||
181             ↪ AreEqual(storedLinkValue.Target, value))
182         {
183             return GetOne();
184         }
185         return GetZero();
186     }
187 }
188
189 [MethodImpl(MethodImplOptions.AggressiveInlining)]
190 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
191 {
192     var constants = Constants;
193     var @break = constants.Break;
194     if (restrictions.Count == 0)
195     {
196         for (var link = GetOne(); LessOrEqualThan(link,
197             ↪ GetHeaderReference().AllocatedLinks); link = Increment(link))
198         {
199             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
200             {
201                 return @break;
202             }
203         }
204         return @break;
205     }
206     var @continue = constants.Continue;
207     var any = constants.Any;
208     var index = restrictions[constants.IndexPart];
209     if (restrictions.Count == 1)
210     {
211         if (AreEqual(index, any))
212         {
213             return Each(handler, GetEmptyList());
214         }
215         if (!Exists(index))
216         {
217             return @continue;
218         }
219         return handler(GetLinkStruct(index));
220     }
221     if (restrictions.Count == 2)
222     {
223         var value = restrictions[1];
224         if (AreEqual(index, any))
225         {
226             if (AreEqual(value, any))
227             {
228                 return Each(handler, GetEmptyList());
229             }
230             if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
231             {
232                 return @break;
233             }
234             return Each(handler, new Link<TLink>(index, any, value));
235         }
236         else
237         {

```

```

237         if (!Exists(index))
238         {
239             return @continue;
240         }
241         if (AreEqual(value, any))
242         {
243             return handler(GetLinkStruct(index));
244         }
245         ref var storedLinkValue = ref GetLinkReference(index);
246         if (AreEqual(storedLinkValue.Source, value) ||
247             AreEqual(storedLinkValue.Target, value))
248         {
249             return handler(GetLinkStruct(index));
250         }
251         return @continue;
252     }
253 }
254 if (restrictions.Count == 3)
255 {
256     var source = restrictions[constants.SourcePart];
257     var target = restrictions[constants.TargetPart];
258     if (AreEqual(index, any))
259     {
260         if (AreEqual(source, any) && AreEqual(target, any))
261         {
262             return Each(handler, GetEmptyList());
263         }
264         else if (AreEqual(source, any))
265         {
266             return TargetsTreeMethods.EachUsage(target, handler);
267         }
268         else if (AreEqual(target, any))
269         {
270             return SourcesTreeMethods.EachUsage(source, handler);
271         }
272         else //if(source != Any && target != Any)
273         {
274             var link = SourcesTreeMethods.Search(source, target);
275             return AreEqual(link, constants.Null) ? @continue :
276                 ↪ handler(GetLinkStruct(link));
277         }
278     }
279     else
280     {
281         if (!Exists(index))
282         {
283             return @continue;
284         }
285         if (AreEqual(source, any) && AreEqual(target, any))
286         {
287             return handler(GetLinkStruct(index));
288         }
289         ref var storedLinkValue = ref GetLinkReference(index);
290         if (!AreEqual(source, any) && !AreEqual(target, any))
291         {
292             if (AreEqual(storedLinkValue.Source, source) &&
293                 AreEqual(storedLinkValue.Target, target))
294             {
295                 return handler(GetLinkStruct(index));
296             }
297             return @continue;
298         }
299         var value = default(TLink);
300         if (AreEqual(source, any))
301         {
302             value = target;
303         }
304         if (AreEqual(target, any))
305         {
306             value = source;
307         }
308         if (AreEqual(storedLinkValue.Source, value) ||
309             AreEqual(storedLinkValue.Target, value))
310         {
311             return handler(GetLinkStruct(index));
312         }
313         return @continue;
314     }
315 }

```

```

314     }
315     throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
316 }
317
318 /// <remarks>
319 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↳ в другом месте (но не в менеджере памяти, а в логике Links)
320 /// </remarks>
321 [MethodImpl(MethodImplOptions.AggressiveInlining)]
322 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
323 {
324     var constants = Constants;
325     var @null = constants.Null;
326     var linkIndex = restrictions[constants.IndexPart];
327     ref var link = ref GetLinkReference(linkIndex);
328     ref var header = ref GetHeaderReference();
329     ref var firstAsSource = ref header.FirstAsSource;
330     ref var firstAsTarget = ref header.FirstAsTarget;
331     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↳ предварительно заполнено нулями
332     if (!AreEqual(link.Source, @null))
333     {
334         SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
335     }
336     if (!AreEqual(link.Target, @null))
337     {
338         TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
339     }
340     link.Source = substitution[constants.SourcePart];
341     link.Target = substitution[constants.TargetPart];
342     if (!AreEqual(link.Source, @null))
343     {
344         SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
345     }
346     if (!AreEqual(link.Target, @null))
347     {
348         TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
349     }
350     return linkIndex;
351 }
352
353 /// <remarks>
354 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
    ↳ пространство
355 /// </remarks>
356 public virtual TLink Create(IList<TLink> restrictions)
357 {
358     ref var header = ref GetHeaderReference();
359     var freeLink = header.FirstFreeLink;
360     if (!AreEqual(freeLink, Constants.Null))
361     {
362         UnusedLinksListMethods.Detach(freeLink);
363     }
364     else
365     {
366         var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
367         if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
368         {
369             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
370         }
371         if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
372         {
373             _memory.ReservedCapacity += _memory.ReservationStep;
374             SetPointers(_memory);
375             header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
    ↳ LinkSizeInBytes);
376         }
377         header.AllocatedLinks = Increment(header.AllocatedLinks);
378         _memory.UsedCapacity += LinkSizeInBytes;
379         freeLink = header.AllocatedLinks;
380     }
381     return freeLink;
382 }
383
384 [MethodImpl(MethodImplOptions.AggressiveInlining)]
385 public virtual void Delete(IList<TLink> restrictions)
386 {
387     ref var header = ref GetHeaderReference();

```

```

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

```

```

463     protected virtual TLink GetZero() => Integer<TLink>.Zero;
464
465     [MethodImpl(MethodImplOptions.AggressiveInlining)]
466     protected virtual bool AreEqual(TLink first, TLink second) =>
467         ↪ EqualityComparer.Equals(first, second);
468
469     [MethodImpl(MethodImplOptions.AggressiveInlining)]
470     protected virtual bool LessThan(TLink first, TLink second) => Comparer.Compare(first,
471         ↪ second) < 0;
472
473     [MethodImpl(MethodImplOptions.AggressiveInlining)]
474     protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
475         ↪ Comparer.Compare(first, second) <= 0;
476
477     [MethodImpl(MethodImplOptions.AggressiveInlining)]
478     protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
479         ↪ Comparer.Compare(first, second) >= 0;
480
481     [MethodImpl(MethodImplOptions.AggressiveInlining)]
482     protected virtual long ConvertToInt64(TLink value) => (Integer<TLink>)value;
483
484     [MethodImpl(MethodImplOptions.AggressiveInlining)]
485     protected virtual TLink ConvertToAddress(long value) => (Integer<TLink>)value;
486
487     [MethodImpl(MethodImplOptions.AggressiveInlining)]
488     protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
489         ↪ second);
490
491     [MethodImpl(MethodImplOptions.AggressiveInlining)]
492     protected virtual TLink Subtract(TLink first, TLink second) =>
493         ↪ Arithmetic<TLink>.Subtract(first, second);
494
495     [MethodImpl(MethodImplOptions.AggressiveInlining)]
496     protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
497
498     [MethodImpl(MethodImplOptions.AggressiveInlining)]
499     protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
500
501     [MethodImpl(MethodImplOptions.AggressiveInlining)]
502     protected virtual IList<TLink> GetEmptyList() => Array.Empty<TLink>();
503
504     #region Disposable
505
506     protected override bool AllowMultipleDisposeCalls => true;
507
508     protected override void Dispose(bool manual, bool wasDisposed)
509     {
510         if (!wasDisposed)
511         {
512             ResetPointers();
513             _memory.DisposeIfPossible();
514         }
515     }
516
517     #endregion
518 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Methods.Lists;
3  using Platform.Numbers;
4  using static System.Runtime.CompilerServices.Unsafe;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
9  {
10     public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
11         ↪ ILinksListMethods<TLink>
12     {
13         private readonly byte* _links;
14         private readonly byte* _header;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public UnusedLinksListMethods(byte* links, byte* header)

```

```

17     {
18         _links = links;
19         _header = header;
20     }
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
24         ↳ AsRef<LinksHeader<TLink>>(_header);
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
28         ↳ AsRef<RawLink<TLink>>(_links + RawLink<TLink>.SizeInBytes * (Integer<TLink>)link);
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected override TLink GetPrevious(TLink element) => GetLinkReference(element).Source;
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected override TLink GetNext(TLink element) => GetLinkReference(element).Target;
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override TLink GetSize() => GetHeaderReference().FreeLinks;
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
47         ↳ element;
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
51         ↳ element;
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override void SetPrevious(TLink element, TLink previous) =>
55         ↳ GetLinkReference(element).Source = previous;
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override void SetNext(TLink element, TLink next) =>
59         ↳ GetLinkReference(element).Target = next;
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
63 }

```

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

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets.ResizableDirectMemory
4  {
5      public interface ILinksListMethods<TLink>
6      {
7          void Detach(TLink freeLink);
8          void AttachAsFirst(TLink link);
9      }
10 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.ResizableDirectMemory
7  {
8      public interface ILinksTreeMethods<TLink>
9      {
10         TLink CountUsages(TLink link);
11         TLink Search(TLink source, TLink target);
12         TLink EachUsage(TLink source, Func<IList<TLink>, TLink> handler);
13         void Detach(ref TLink firstAsSource, TLink linkIndex);
14         void Attach(ref TLink firstAsSource, TLink linkIndex);
15     }
16 }

```

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

```

1 using Platform.Unsafe;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.ResizableDirectMemory
6 {
7     public struct LinksHeader<TLink>
8     {
9         public static readonly long SizeInBytes = Structure<LinksHeader<TLink>>.Size;
10
11         public TLink AllocatedLinks;
12         public TLink ReservedLinks;
13         public TLink FreeLinks;
14         public TLink FirstFreeLink;
15         public TLink FirstAsSource;
16         public TLink FirstAsTarget;
17         public TLink LastFreeLink;
18         public TLink Reserved8;
19     }
20 }

```

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

```

1 using Platform.Unsafe;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.ResizableDirectMemory
6 {
7     public struct RawLink<TLink>
8     {
9         public static readonly long SizeInBytes = Structure<RawLink<TLink>>.Size;
10
11         public TLink Source;
12         public TLink Target;
13         public TLink LeftAsSource;
14         public TLink RightAsSource;
15         public TLink SizeAsSource;
16         public TLink LeftAsTarget;
17         public TLink RightAsTarget;
18         public TLink SizeAsTarget;
19     }
20 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Data.Doublets.ResizableDirectMemory.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.ResizableDirectMemory.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         public 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             [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         }
38     }
39 }

```



```

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) => unchecked((value & 4294967264UL)
88 ↪ >> 5);
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
92 ↪ unchecked(storedValue & 31UL | (size & 134217727UL) << 5);
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 protected override bool GetLeftIsChildValue(ulong value) => unchecked((value & 16UL) >>
96 ↪ 4 == 1UL);
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
100 ↪ storedValue = unchecked(storedValue & 4294967279UL | (As<bool, byte>(ref value) &
101 ↪ 1UL) << 4);
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 protected override bool GetRightIsChildValue(ulong value) => unchecked((value & 8UL) >>
105 ↪ 3 == 1UL);
106
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]
108 protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>
109 ↪ storedValue = unchecked(storedValue & 4294967287UL | (As<bool, byte>(ref value) &
110 ↪ 1UL) << 3);

```

```

100 [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 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Data.Doublets.ResizableDirectMemory.Generic;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.ResizableDirectMemory.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         public 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             [MethodImpl(MethodImplOptions.AggressiveInlining)]
22             protected override ulong GetZero() => 0UL;
23
24             [MethodImpl(MethodImplOptions.AggressiveInlining)]
25             protected override bool EqualToZero(ulong value) => value == 0UL;
26
27             [MethodImpl(MethodImplOptions.AggressiveInlining)]
28             protected override bool AreEqual(ulong first, ulong second) => first == second;
29
30             [MethodImpl(MethodImplOptions.AggressiveInlining)]
31             protected override bool GreaterThanZero(ulong value) => value > 0UL;
32
33             [MethodImpl(MethodImplOptions.AggressiveInlining)]
34             protected override bool GreaterThan(ulong first, ulong second) => first > second;
35
36             [MethodImpl(MethodImplOptions.AggressiveInlining)]
37             protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
38
39             [MethodImpl(MethodImplOptions.AggressiveInlining)]
40             protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
41             ↳ always true for ulong
42
43             [MethodImpl(MethodImplOptions.AggressiveInlining)]
44             protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
45             ↳ always >= 0 for ulong
46
47             [MethodImpl(MethodImplOptions.AggressiveInlining)]
48             protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
49
50             [MethodImpl(MethodImplOptions.AggressiveInlining)]
51             protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
52             ↳ for ulong
53
54             [MethodImpl(MethodImplOptions.AggressiveInlining)]
55             protected override bool LessThan(ulong first, ulong second) => first < second;
56
57             [MethodImpl(MethodImplOptions.AggressiveInlining)]
58             protected override ulong Increment(ulong value) => ++value;
59
60             [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

57     protected override ulong Decrement(ulong value) => --value;
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override ulong Add(ulong first, ulong second) => first + second;
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected override ulong Subtract(ulong first, ulong second) => first - second;
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
67     {
68         ref var firstLink = ref Links[first];
69         ref var secondLink = ref Links[second];
70         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
71             ↪ secondLink.Source, secondLink.Target);
72     }
73
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
76     {
77         ref var firstLink = ref Links[first];
78         ref var secondLink = ref Links[second];
79         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
80             ↪ secondLink.Source, secondLink.Target);
81     }
82
83     [MethodImpl(MethodImplOptions.AggressiveInlining)]
84     protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
85
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
88 }

```

1.48 ./Platform.Data.Doublets/ResizableDirectMemory/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.ResizableDirectMemory.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)]

```

```

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

```

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

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
6  {
7      public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
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;

```

```

16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
24         ↳ left;
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
28         ↳ right;
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     protected override ulong GetSize(ulong node) => Links[node].SizeAsSource;
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsSource =
35         ↳ size;
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override ulong GetTreeRoot() => Header->FirstAsSource;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
45         ↳ ulong secondSource, ulong secondTarget)
46         => firstSource < secondSource || firstSource == secondSource && firstTarget <
47         ↳ secondTarget;
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
51         ↳ ulong secondSource, ulong secondTarget)
52         => firstSource > secondSource || firstSource == secondSource && firstTarget >
53         ↳ secondTarget;
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override void ClearNode(ulong node)
57     {
58         ref var link = ref Links[node];
59         link.LeftAsSource = OUL;
60         link.RightAsSource = OUL;
61         link.SizeAsSource = OUL;
62     }
63 }

```

1.50 ./Platform.Data.Doublets/ResizableDirectMemory/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.ResizableDirectMemory.Specific
6  {
7      public unsafe class UInt64LinksTargetsAvlBalancedTreeMethods :
8          ↳ UInt64LinksAvlBalancedTreeMethodsBase
9      {
10         public UInt64LinksTargetsAvlBalancedTreeMethods(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].LeftAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20             ↳ Links[node].RightAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
27
28     }
29 }

```

```

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->FirstAsTarget;
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
    ↪ ulong secondSource, ulong secondTarget)
61     => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
    ↪ secondSource;
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
    ↪ ulong secondSource, ulong secondTarget)
65     => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
    ↪ secondSource;
66
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     protected override void ClearNode(ulong node)
69     {
70         ref var link = ref Links[node];
71         link.LeftAsTarget = OUL;
72         link.RightAsTarget = OUL;
73         link.SizeAsTarget = OUL;
74     }
75 }
76 }

```

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

```

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

```

```

9      public UInt64LinksTargetsSizeBalancedTreeMethods(LinksConstants<ulong> constants,
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->FirstAsTarget;
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 }

```

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

```

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

```

```

18 [MethodImpl(MethodImplOptions.AggressiveInlining)]
19 public UInt64ResizableDirectMemoryLinks(string address) : this(address,
20     ↳ DefaultLinksSizeStep) { }
21
22 /// <summary>
23 /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
24     ↳ минимальным шагом расширения базы данных.
25 /// </summary>
26 /// <param name="address">Полный путь к файлу базы данных.</param>
27 /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
28     ↳ байтах.</param>
29 [MethodImpl(MethodImplOptions.AggressiveInlining)]
30 public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
31     ↳ this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
32     ↳ memoryReservationStep) { }
33
34 [MethodImpl(MethodImplOptions.AggressiveInlining)]
35 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
36     ↳ DefaultLinksSizeStep) { }
37
38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
40     ↳ memoryReservationStep) : this(memory, memoryReservationStep,
41     ↳ Default<LinksConstants<ulong>>.Instance, true) { }
42
43 [MethodImpl(MethodImplOptions.AggressiveInlining)]
44 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
45     ↳ memoryReservationStep, LinksConstants<ulong> constants, bool useAvlBasedIndex) :
46     ↳ base(memory, memoryReservationStep, constants)
47 {
48     if (useAvlBasedIndex)
49     {
50         _createSourceTreeMethods = () => new
51             ↳ UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
52         _createTargetTreeMethods = () => new
53             ↳ UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
54     }
55     else
56     {
57         _createSourceTreeMethods = () => new
58             ↳ UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
59         _createTargetTreeMethods = () => new
60             ↳ UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
61     }
62     Init(memory, memoryReservationStep);
63 }
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override void SetPointers(IResizableDirectMemory memory)
67 {
68     _header = (LinksHeader<ulong>*)memory.Pointer;
69     _links = (RawLink<ulong>*)memory.Pointer;
70     SourcesTreeMethods = _createSourceTreeMethods();
71     TargetsTreeMethods = _createTargetTreeMethods();
72     UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
73 }
74
75 [MethodImpl(MethodImplOptions.AggressiveInlining)]
76 protected override void ResetPointers()
77 {
78     base.ResetPointers();
79     _links = null;
80     _header = null;
81 }
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
88     ↳ _links[linkIndex];
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 protected override bool AreEqual(ulong first, ulong second) => first == second;
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

80     protected override bool LessThan(ulong first, ulong second) => first < second;
81
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     protected override bool GreaterThan(ulong first, ulong second) => first > second;
87
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
90
91     [MethodImpl(MethodImplOptions.AggressiveInlining)]
92     protected override ulong GetZero() => 0UL;
93
94     [MethodImpl(MethodImplOptions.AggressiveInlining)]
95     protected override ulong GetOne() => 1UL;
96
97     [MethodImpl(MethodImplOptions.AggressiveInlining)]
98     protected override long ConvertToUInt64(ulong value) => (long)value;
99
100    [MethodImpl(MethodImplOptions.AggressiveInlining)]
101    protected override ulong ConvertToAddress(long value) => (ulong)value;
102
103    [MethodImpl(MethodImplOptions.AggressiveInlining)]
104    protected override ulong Add(ulong first, ulong second) => first + second;
105
106    [MethodImpl(MethodImplOptions.AggressiveInlining)]
107    protected override ulong Subtract(ulong first, ulong second) => first - second;
108
109    [MethodImpl(MethodImplOptions.AggressiveInlining)]
110    protected override ulong Increment(ulong link) => ++link;
111
112    [MethodImpl(MethodImplOptions.AggressiveInlining)]
113    protected override ulong Decrement(ulong link) => --link;
114
115    [MethodImpl(MethodImplOptions.AggressiveInlining)]
116    protected override IList<ulong> GetEmptyList() => new ulong[0];
117 }
118 }

```

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

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences
7  {
8      public static class ArrayExtensions
9      {
10         public static IList<TLink> ConvertToRestrictionsValues<TLink>(this TLink[] array)

```

```

11     {
12         var restrictions = new TLink[array.Length + 1];
13         Array.Copy(array, 0, restrictions, 1, array.Length);
14         return restrictions;
15     }
16 }
17 }

```

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

```

1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Converters
6  {
7      public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
8      {
9          public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }
10
11         public override TLink Convert(ICollection<TLink> sequence)
12         {
13             var length = sequence.Count;
14             if (length < 1)
15             {
16                 return default;
17             }
18             if (length == 1)
19             {
20                 return sequence[0];
21             }
22             // Make copy of next layer
23             if (length > 2)
24             {
25                 // TODO: Try to use stackalloc (which at the moment is not working with
26                 // ↪ generics) but will be possible with Sigil
27                 var halvedSequence = new TLink[(length / 2) + (length % 2)];
28                 HalveSequence(halvedSequence, sequence, length);
29                 sequence = halvedSequence;
30                 length = halvedSequence.Length;
31             }
32             // Keep creating layer after layer
33             while (length > 2)
34             {
35                 HalveSequence(sequence, sequence, length);
36                 length = (length / 2) + (length % 2);
37             }
38             return Links.GetOrCreate(sequence[0], sequence[1]);
39         }
40
41         private void HalveSequence(ICollection<TLink> destination, ICollection<TLink> source, int length)
42         {
43             var loopedLength = length - (length % 2);
44             for (var i = 0; i < loopedLength; i += 2)
45             {
46                 destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
47             }
48             if (length > loopedLength)
49             {
50                 destination[length / 2] = source[length - 1];
51             }
52         }
53     }
54 }

```

1.56 ./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 readonly IConverter<IList<TLink>, TLink> _baseConverter;
31      private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
32      private readonly TLink _minFrequencyToCompress;
33      private readonly bool _doInitialFrequenciesIncrement;
34      private Doublet<TLink> _maxDoublet;
35      private LinkFrequency<TLink> _maxDoubletData;
36
37      private struct HalfDoublet
38      {
39          public TLink Element;
40          public LinkFrequency<TLink> DoubletData;
41
42          public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
43          {
44              Element = element;
45              DoubletData = doubletData;
46          }
47
48          public override string ToString() => $"{Element}: ({DoubletData})";
49      }
50
51      public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
52      ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
53      : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One, true)
54      {
55      }
56
57      public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
58      ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
59      ↪ doInitialFrequenciesIncrement)
60      : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One,
61      ↪ doInitialFrequenciesIncrement)
62      {
63      }
64
65      public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
66      ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
67      ↪ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
68      : base(links)
69      {
70          _baseConverter = baseConverter;
71          _doubletFrequenciesCache = doubletFrequenciesCache;
72          if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
73          {
74              minFrequencyToCompress = Integer<TLink>.One;
75          }
76          _minFrequencyToCompress = minFrequencyToCompress;
77          _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
78          ResetMaxDoublet();
79      }
80
81      public override TLink Convert(IList<TLink> source) =>
82      ↪ _baseConverter.Convert(Compress(source));
83
84      /// <remarks>
85      /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
86      /// Faster version (doublets' frequencies dictionary is not recreated).
87      /// </remarks>
88      private IList<TLink> Compress(IList<TLink> sequence)
89      {
90          if (sequence.IsNullOrEmpty())
91          {
92              return null;
93          }
94          if (sequence.Count == 1)

```

```

{
    return sequence;
}
if (sequence.Count == 2)
{
    return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
}
// TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
var copy = new HalfDoublet[sequence.Count];
Doublet<TLink> doublet = default;
for (var i = 1; i < sequence.Count; i++)
{
    doublet.Source = sequence[i - 1];
    doublet.Target = sequence[i];
    LinkFrequency<TLink> data;
    if (_doInitialFrequenciesIncrement)
    {
        data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
    }
    else
    {
        data = _doubletFrequenciesCache.GetFrequency(ref doublet);
        if (data == null)
        {
            throw new NotSupportedException("If you ask not to increment
            ↪ frequencies, it is expected that all frequencies for the sequence
            ↪ are prepared.");
        }
    }
    copy[i - 1].Element = sequence[i - 1];
    copy[i - 1].DoubletData = data;
    UpdateMaxDoublet(ref doublet, data);
}
copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
{
    var newLength = ReplaceDoublets(copy);
    sequence = new TLink[newLength];
    for (int i = 0; i < newLength; i++)
    {
        sequence[i] = copy[i].Element;
    }
}
return sequence;
}

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

```

```

157     }
158     if (r < oldLengthMinusTwo)
159     {
160         var next = copy[r + 2].Element;
161         copy[r + 1].DoubletData.DecrementFrequency();
162         copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
            ↪ xDoubletReplacementLink,
            ↪ next);
163     }
164     copy[w++].Element = maxDoubletReplacementLink;
165     r++;
166     newLength--;
167 }
168 else
169 {
170     copy[w++] = copy[r];
171 }
172 }
173 if (w < newLength)
174 {
175     copy[w] = copy[r];
176 }
177 oldLength = newLength;
178 ResetMaxDoublet();
179 UpdateMaxDoublet(copy, newLength);
180 }
181 return newLength;
182 }
183
184 [MethodImpl(MethodImplOptions.AggressiveInlining)]
185 private void ResetMaxDoublet()
186 {
187     _maxDoublet = new Doublet<TLink>();
188     _maxDoubletData = new LinkFrequency<TLink>();
189 }
190
191 [MethodImpl(MethodImplOptions.AggressiveInlining)]
192 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
193 {
194     Doublet<TLink> doublet = default;
195     for (var i = 1; i < length; i++)
196     {
197         doublet.Source = copy[i - 1].Element;
198         doublet.Target = copy[i].Element;
199         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
200     }
201 }
202
203 [MethodImpl(MethodImplOptions.AggressiveInlining)]
204 private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
205 {
206     var frequency = data.Frequency;
207     var maxFrequency = _maxDoubletData.Frequency;
208     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
209     ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
210     ↪ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
211     ↪ _maxDoublet.Target)))
212     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
213     ↪ (_comparer.Compare(maxFrequency, frequency) < 0 ||
214     ↪ (_equalityComparer.Equals(maxFrequency, frequency) &&
215     ↪ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
216     ↪ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
217     ↪ better stability and better compression on sequent data and even on random
218     ↪ numbers data (but gives collisions anyway) */
219     {
220         _maxDoublet = doublet;
221         _maxDoubletData = data;
222     }
223 }
224 }
225 }
226 }
227 }

```

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

```

1 using System.Collections.Generic;
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.Converters

```

```

7 {
8     public abstract class LinksListToSequenceConverterBase<TLink> : IConverter<IList<TLink>,
        ↳ TLink>
9     {
10         protected readonly ILinks<TLink> Links;
11         public LinksListToSequenceConverterBase(ILinks<TLink> links) => Links = links;
12         public abstract TLink Convert(IList<TLink> source);
13     }
14 }

```

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

```

1 using System.Collections.Generic;
2 using System.Linq;
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 OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
12         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
13
14         private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
15
16         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
            ↳ sequenceToItsLocalElementLevelsConverter) : base(links)
17         => _sequenceToItsLocalElementLevelsConverter =
            ↳ sequenceToItsLocalElementLevelsConverter;
18
19         public override TLink Convert(IList<TLink> sequence)
20         {
21             var length = sequence.Count;
22             if (length == 1)
23             {
24                 return sequence[0];
25             }
26             var links = Links;
27             if (length == 2)
28             {
29                 return links.GetOrCreate(sequence[0], sequence[1]);
30             }
31             sequence = sequence.ToArray();
32             var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
33             while (length > 2)
34             {
35                 var levelRepeat = 1;
36                 var currentLevel = levels[0];
37                 var previousLevel = levels[0];
38                 var skipOnce = false;
39                 var w = 0;
40                 for (var i = 1; i < length; i++)
41                 {
42                     if (_equalityComparer.Equals(currentLevel, levels[i]))
43                     {
44                         levelRepeat++;
45                         skipOnce = false;
46                         if (levelRepeat == 2)
47                         {
48                             sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
49                             var newLevel = i >= length - 1 ?
35                             GetPreviousLowerThanCurrentOrCurrent(previousLevel,
36                                 ↳ currentLevel) :
51                             i < 2 ?
52                             GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
53                             GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
35                                 ↳ currentLevel, levels[i + 1]);
54                             levels[w] = newLevel;
55                             previousLevel = currentLevel;
56                             w++;
57                             levelRepeat = 0;
58                             skipOnce = true;
59                         }
60                         else if (i == length - 1)
61                         {
62                             sequence[w] = sequence[i];
63                             levels[w] = levels[i];
64                             w++;

```

```

65     }
66     }
67     else
68     {
69         currentLevel = levels[i];
70         levelRepeat = 1;
71         if (skipOnce)
72         {
73             skipOnce = false;
74         }
75         else
76         {
77             sequence[w] = sequence[i - 1];
78             levels[w] = levels[i - 1];
79             previousLevel = levels[w];
80             w++;
81         }
82         if (i == length - 1)
83         {
84             sequence[w] = sequence[i];
85             levels[w] = levels[i];
86             w++;
87         }
88     }
89     }
90     length = w;
91 }
92 return links.GetOrCreate(sequence[0], sequence[1]);
93 }
94
95 private static TLink GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink previous, TLink
↪ current, TLink next)
96 {
97     return _comparer.Compare(previous, next) > 0
98         ? _comparer.Compare(previous, current) < 0 ? previous : current
99         : _comparer.Compare(next, current) < 0 ? next : current;
100 }
101
102 private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
↪ _comparer.Compare(next, current) < 0 ? next : current;
103
104 private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
↪ => _comparer.Compare(previous, current) < 0 ? previous : current;
105 }
106 }

```

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

```

1 using System.Collections.Generic;
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.Converters
7 {
8     public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
↪ IConverter<IList<TLink>>
9     {
10         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
11
12         private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
13
14         public SequenceToItsLocalElementLevelsConverter(IList<TLink> links,
↪ IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
↪ => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
15
16         public IList<TLink> Convert(IList<TLink> sequence)
17         {
18             var levels = new TLink[sequence.Count];
19             levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
20             for (var i = 1; i < sequence.Count - 1; i++)
21             {
22                 var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
23                 var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
24                 levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
25             }
26             levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
↪ sequence[sequence.Count - 1]);
27             return levels;
28         }
29     }

```

```

29
30     public TLink GetFrequencyNumber(TLink source, TLink target) =>
        ↪ _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
31 }
32 }

```

1.60 ./Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.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.CriterionMatchers
6 {
7     public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
        ↪ ICriterionMatcher<TLink>
8     {
9         public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
10        public bool IsMatched(TLink argument) => Links.IsPartialPoint(argument);
11    }
12 }

```

1.61 ./Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs

```

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

```

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

```

1 using System.Collections.Generic;
2 using Platform.Collections.Stacks;
3 using Platform.Data.Doublets.Sequences.HeightProviders;
4 using Platform.Data.Sequences;
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 DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
        ↪ ISequenceAppender<TLink>
11    {
12        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;
13
14        private readonly IStack<TLink> _stack;
15        private readonly ISequenceHeightProvider<TLink> _heightProvider;
16
17        public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
        ↪ ISequenceHeightProvider<TLink> heightProvider)
        : base(links)
18        {
19            _stack = stack;
20            _heightProvider = heightProvider;
21        }
22
23        public TLink Append(TLink sequence, TLink appendant)
24        {
25            var cursor = sequence;
26

```



```

27         while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
28         {
29             var source = Links.GetSource(cursor);
30             var target = Links.GetTarget(cursor);
31             if (_equalityComparer.Equals(_heightProvider.Get(source),
32                 ↪ _heightProvider.Get(target)))
33             {
34                 break;
35             }
36             else
37             {
38                 _stack.Push(source);
39                 cursor = target;
40             }
41         }
42         var left = cursor;
43         var right = appendant;
44         while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
45         {
46             right = Links.GetOrCreate(left, right);
47             left = cursor;
48         }
49         return Links.GetOrCreate(left, right);
50     }
51 }

```

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

```

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

```

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

```

1 using System;
2 using System.Linq;
3 using System.Collections.Generic;
4 using Platform.Interfaces;
5 using Platform.Collections;
6 using Platform.Collections.Lists;
7 using Platform.Collections.Segments;
8 using Platform.Collections.Segments.Walkers;
9 using Platform.Singletons;
10 using Platform.Numbers;
11 using Platform.Data.Doublets.Unicode;
12
13 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14
15 namespace Platform.Data.Doublets.Sequences
16 {
17     public class DuplicateSegmentsProvider<TLink> :
18         ↪ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
19         ↪ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
20     {
21         private readonly IList<TLink> _links;
22         private readonly IList<TLink> _sequences;
23         private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
24         private BitString _visited;
25
26         private class ItemEqualityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
27             ↪ IList<TLink>>>
28         {
29             private readonly IListEqualityComparer<TLink> _listComparer;
30             public ItemEqualityComparer() => _listComparer =
31                 ↪ Default<IListEqualityComparer<TLink>>.Instance;
32         }
33     }
34 }

```

```

28     public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
    ↪     KeyValuePair<IList<TLink>, IList<TLink>> right) =>
    ↪     _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
    ↪     right.Value);
29     public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
    ↪     (_listComparer.GetHashCode(pair.Key),
    ↪     _listComparer.GetHashCode(pair.Value)).GetHashCode();
30 }
31
32 private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
33 {
34     private readonly IListComparer<TLink> _listComparer;
35
36     public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
37
38     public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
    ↪     KeyValuePair<IList<TLink>, IList<TLink>> right)
39     {
40         var intermediateResult = _listComparer.Compare(left.Key, right.Key);
41         if (intermediateResult == 0)
42         {
43             intermediateResult = _listComparer.Compare(left.Value, right.Value);
44         }
45         return intermediateResult;
46     }
47 }
48
49 public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
50     : base(minimumStringSegmentLength: 2)
51 {
52     _links = links;
53     _sequences = sequences;
54 }
55
56 public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
57 {
58     _groups = new HashSet<KeyValuePair<IList<TLink>,
    ↪     IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
59     var count = _links.Count();
60     _visited = new BitString((long)(Integer<TLink>)count + 1);
61     _links.Each(link =>
62     {
63         var linkIndex = _links.GetIndex(link);
64         var linkBitIndex = (long)(Integer<TLink>)linkIndex;
65         if (!_visited.Get(linkBitIndex))
66         {
67             var sequenceElements = new List<TLink>();
68             var filler = new ListFiller<TLink, TLink>(sequenceElements,
    ↪             _sequences.Constants.Break);
69             _sequences.Each(filler.AddAllValuesAndReturnConstant, new
    ↪             LinkAddress<TLink>(linkIndex));
70             if (sequenceElements.Count > 2)
71             {
72                 WalkAll(sequenceElements);
73             }
74         }
75         return _links.Constants.Continue;
76     });
77     var resultList = _groups.ToList();
78     var comparer = Default<ItemComparer>.Instance;
79     resultList.Sort(comparer);
80     #if DEBUG
81     foreach (var item in resultList)
82     {
83         PrintDuplicates(item);
84     }
85     #endif
86     return resultList;
87 }
88
89 protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
    ↪     length) => new Segment<TLink>(elements, offset, length);
90
91 protected override void OnDuplicateFound(Segment<TLink> segment)
92 {
93     var duplicates = CollectDuplicatesForSegment(segment);
94     if (duplicates.Count > 1)
95     {

```

```

96         _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
97             ↪ duplicates));
98     }
99 }
100 private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
101 {
102     var duplicates = new List<TLink>();
103     var readAsElement = new HashSet<TLink>();
104     var restrictions = segment.ConvertToRestrictionsValues();
105     restrictions[0] = _sequences.Constants.Any;
106     _sequences.Each(sequence =>
107     {
108         var sequenceIndex = sequence[_sequences.Constants.IndexPart];
109         duplicates.Add(sequenceIndex);
110         readAsElement.Add(sequenceIndex);
111         return _sequences.Constants.Continue;
112     }, restrictions);
113     if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
114     {
115         return new List<TLink>();
116     }
117     foreach (var duplicate in duplicates)
118     {
119         var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
120         _visited.Set(duplicateBitIndex);
121     }
122     if (_sequences is Sequences sequencesExperiments)
123     {
124         var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
125             ↪ ashSet<ulong>)(object)readAsElement,
126             ↪ (IList<ulong>)segment);
127         foreach (var partiallyMatchedSequence in partiallyMatched)
128         {
129             TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
130             duplicates.Add(sequenceIndex);
131         }
132     }
133     duplicates.Sort();
134     return duplicates;
135 }
136 private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
137 {
138     if (!(_links is ILinks<ulong> ulongLinks))
139     {
140         return;
141     }
142     var duplicatesKey = duplicatesItem.Key;
143     var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
144     Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
145     var duplicatesList = duplicatesItem.Value;
146     for (int i = 0; i < duplicatesList.Count; i++)
147     {
148         ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
149         var formattedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
150             ↪ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
151             ↪ UnicodeMap.IsCharLink(link.Index) ?
152             ↪ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
153         Console.WriteLine(formattedSequenceStructure);
154         var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
155             ↪ ulongLinks);
156         Console.WriteLine(sequenceString);
157     }
158     Console.WriteLine();
159 }
160 }

```

1.65 ./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 readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
23         private readonly ICounter<TLink, TLink> _frequencyCounter;
24
25         public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
26             : base(links)
27         {
28             _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
29                 ↪ DoubletComparer<TLink>.Default);
30             _frequencyCounter = frequencyCounter;
31
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
34             {
35                 var doublet = new Doublet<TLink>(source, target);
36                 return GetFrequency(ref doublet);
37             }
38
39             [MethodImpl(MethodImplOptions.AggressiveInlining)]
40             public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
41             {
42                 _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
43                 return data;
44             }
45
46             public void IncrementFrequencies(IList<TLink> sequence)
47             {
48                 for (var i = 1; i < sequence.Count; i++)
49                 {
50                     IncrementFrequency(sequence[i - 1], sequence[i]);
51                 }
52             }
53
54             [MethodImpl(MethodImplOptions.AggressiveInlining)]
55             public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
56             {
57                 var doublet = new Doublet<TLink>(source, target);
58                 return IncrementFrequency(ref doublet);
59             }
60
61             public void PrintFrequencies(IList<TLink> sequence)
62             {
63                 for (var i = 1; i < sequence.Count; i++)
64                 {
65                     PrintFrequency(sequence[i - 1], sequence[i]);
66                 }
67             }
68
69             public void PrintFrequency(TLink source, TLink target)
70             {
71                 var number = GetFrequency(source, target).Frequency;
72                 Console.WriteLine("{0},{1}) - {2}", source, target, number);
73             }
74
75             [MethodImpl(MethodImplOptions.AggressiveInlining)]
76             public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
77             {
78                 if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
79                 {
80                     data.IncrementFrequency();
81                 }
82                 else
83                 {
84                     var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
85                     data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
86                     if (!_equalityComparer.Equals(link, default))

```

```

85         {
86             data.Frequency = Arithmetic.Add(data.Frequency,
87                 ↪ _frequencyCounter.Count(link));
88         }
89         _doubletsCache.Add(doublet, data);
90     }
91     return data;
92 }
93 public void ValidateFrequencies()
94 {
95     foreach (var entry in _doubletsCache)
96     {
97         var value = entry.Value;
98         var linkIndex = value.Link;
99         if (!_equalityComparer.Equals(linkIndex, default))
100         {
101             var frequency = value.Frequency;
102             var count = _frequencyCounter.Count(linkIndex);
103             // TODO: Why `frequency` always greater than `count` by 1?
104             if (((_comparer.Compare(frequency, count) > 0) &&
105                 ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count),
106                     ↪ Integer<TLink>.One) > 0))
107                 || ((_comparer.Compare(count, frequency) > 0) &&
108                     ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency),
109                         ↪ Integer<TLink>.One) > 0)))
110             {
111                 throw new InvalidOperationException("Frequencies validation failed.");
112             }
113             //else
114             //{
115             //    if (value.Frequency > 0)
116             //    {
117             //        var frequency = value.Frequency;
118             //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
119             //        var count = _countLinkFrequency(linkIndex);
120             //        if ((frequency > count && frequency - count > 1) || (count > frequency
121             //            ↪ && count - frequency > 1))
122             //            throw new Exception("Frequencies validation failed.");
123             //    }
124             //}
125     }
126 }
127 }
128 }
129 }

```

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

```

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

```
1 using Platform.Converters;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
6 {
7     public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
8         ↳ IConverter<Doublet<TLink>, TLink>
9     {
10         private readonly LinkFrequenciesCache<TLink> _cache;
11         public
12         ↳ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
13         ↳ cache) => _cache = cache;
14         public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
15     }
16 }
```

1.68 ./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.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.Frequencies.Counters
6 {
7     public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
8         ↳ SequenceSymbolFrequencyOneOffCounter<TLink>
9     {
10         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
11
12         public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
13         ↳ ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
14         : base(links, sequenceLink, symbol)
15         => _markedSequenceMatcher = markedSequenceMatcher;
16
17         public override TLink Count()
18         {
19             if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
20             {
21                 return default;
22             }
23             return base.Count();
24         }
25     }
26 }
```

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

```
1 using System.Collections.Generic;
2 using Platform.Interfaces;
3 using Platform.Numbers;
4 using Platform.Data.Sequences;
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 SequenceSymbolFrequencyOneOffCounter<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 _sequenceLink;
18         protected readonly TLink _symbol;
19         protected TLink _total;
20
21         public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
22         ↳ TLink symbol)
23         {
24             _links = links;
25             _sequenceLink = sequenceLink;
26             _symbol = symbol;
27             _total = default;
28         }
29
30         public virtual TLink Count()
31         {
32             if (_comparer.Compare(_total, default) > 0)
33             {
34                 // ...
35             }
36         }
37     }
38 }
```

```

32         return _total;
33     }
34     StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
35         ↪ IsElement, VisitElement);
36     return _total;
37 }
38 private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
39     ↪ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
40     ↪ IsPartialPoint
41
42 private bool VisitElement(TLink element)
43 {
44     if (_equalityComparer.Equals(element, _symbol))
45     {
46         _total = Arithmetic.Increment(_total);
47     }
48     return true;
49 }

```

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

```

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.Frequencies.Counters
6 {
7     public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
8     {
9         private readonly ILinks<TLink> _links;
10        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
11
12        public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
13            ↪ ICriterionMatcher<TLink> markedSequenceMatcher)
14        {
15            _links = links;
16            _markedSequenceMatcher = markedSequenceMatcher;
17        }
18
19        public TLink Count(TLink argument) => new
20            ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
21            ↪ _markedSequenceMatcher, argument).Count();
22    }
23 }

```

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

```

1 using Platform.Interfaces;
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.Counters
7 {
8     public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
9         ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>
10    {
11        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
12
13        public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
14            ↪ ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
15            : base(links, symbol)
16            => _markedSequenceMatcher = markedSequenceMatcher;
17
18        protected override void CountSequenceSymbolFrequency(TLink link)
19        {
20            var symbolFrequencyCounter = new
21                ↪ MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
22                ↪ _markedSequenceMatcher, link, _symbol);
23            _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
24        }
25    }
26 }

```

1.72 ./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.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.Frequencies.Counters
6 {
7     public class TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
8     {
9         private readonly ILinks<TLink> _links;
10        public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;
11        public TLink Count(TLink symbol) => new
12            ↳ TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
13    }

```

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

```

1 using System.Collections.Generic;
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 TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
10    {
11        private static readonly EqualityComparer<TLink> _equalityComparer =
12            ↳ EqualityComparer<TLink>.Default;
13        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
14
15        protected readonly ILinks<TLink> _links;
16        protected readonly TLink _symbol;
17        protected readonly HashSet<TLink> _visits;
18        protected TLink _total;
19
20        public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
21        {
22            _links = links;
23            _symbol = symbol;
24            _visits = new HashSet<TLink>();
25            _total = default;
26        }
27
28        public TLink Count()
29        {
30            if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
31            {
32                return _total;
33            }
34            CountCore(_symbol);
35            return _total;
36        }
37
38        private void CountCore(TLink link)
39        {
40            var any = _links.Constants.Any;
41            if (_equalityComparer.Equals(_links.Count(any, link), default))
42            {
43                CountSequenceSymbolFrequency(link);
44            }
45            else
46            {
47                _links.Each(EachElementHandler, any, link);
48            }
49        }
50
51        protected virtual void CountSequenceSymbolFrequency(TLink link)
52        {
53            var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
54                ↳ link, _symbol);
55            _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
56        }
57
58        private TLink EachElementHandler(IList<TLink> doublet)
59        {
60            var constants = _links.Constants;
61            var doubletIndex = doublet[constants.IndexPart];
62            if (_visits.Add(doubletIndex))
63            {
64                CountCore(doubletIndex);
65            }
66            return constants.Continue;
67        }
68    }

```



```

66     }
67 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
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.HeightProviders
8  {
9      public class CachedSequenceHeightProvider<TLink> : LinksOperatorBase<TLink>,
10         ↳ 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         public CachedSequenceHeightProvider(
22             ILinks<TLink> links,
23             ISequenceHeightProvider<TLink> baseHeightProvider,
24             IConverter<TLink> addressToUnaryNumberConverter,
25             IConverter<TLink> unaryNumberToAddressConverter,
26             TLink heightPropertyMarker,
27             IProperties<TLink, TLink, TLink> propertyOperator)
28             : base(links)
29         {
30             _heightPropertyMarker = heightPropertyMarker;
31             _baseHeightProvider = baseHeightProvider;
32             _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
33             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
34             _propertyOperator = propertyOperator;
35         }
36
37         public TLink Get(TLink sequence)
38         {
39             TLink height;
40             var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
41             if (_equalityComparer.Equals(heightValue, default))
42             {
43                 height = _baseHeightProvider.Get(sequence);
44                 heightValue = _addressToUnaryNumberConverter.Convert(height);
45                 _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
46             }
47             else
48             {
49                 height = _unaryNumberToAddressConverter.Convert(heightValue);
50             }
51             return height;
52         }
53     }
54 }

```

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

```

1  using Platform.Interfaces;
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.HeightProviders
7  {
8      public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
9         ↳ ISequenceHeightProvider<TLink>
10     {
11         private readonly ICriterionMatcher<TLink> _elementMatcher;
12
13         public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
14             ↳ elementMatcher) : base(links) => _elementMatcher = elementMatcher;
15
16         public TLink Get(TLink sequence)
17         {
18             var height = default(TLink);
19             var pairOrElement = sequence;
20             while (!_elementMatcher.IsMatched(pairOrElement))
21             {

```

```

20         pairOrElement = Links.GetTarget(pairOrElement);
21         height = Arithmetic.Increment(height);
22     }
23     return height;
24 }
25 }
26 }

```

1.76 ./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 }

```

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

```

1  using Platform.Collections;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences
7  {
8      public static class IListExtensions
9      {
10         public static TLink[] ExtractValues<TLink>(this IList<TLink> restrictions)
11         {
12             if(restrictions.IsNullOrEmpty() || restrictions.Count == 1)
13             {
14                 return new TLink[0];
15             }
16             var values = new TLink[restrictions.Count - 1];
17             for (int i = 1, j = 0; i < restrictions.Count; i++, j++)
18             {
19                 values[j] = restrictions[i];
20             }
21             return values;
22         }
23
24         public static IList<TLink> ConvertToRestrictionsValues<TLink>(this IList<TLink> list)
25         {
26             var restrictions = new TLink[list.Count + 1];
27             for (int i = 0, j = 1; i < list.Count; i++, j++)
28             {
29                 restrictions[j] = list[i];
30             }
31             return restrictions;
32         }
33     }
34 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
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 CachedFrequencyIncrementingSequenceIndex<TLink> : ISequenceIndex<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ⇨ EqualityComparer<TLink>.Default;
12
13         private readonly LinkFrequenciesCache<TLink> _cache;
14
15         public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
16             ⇨ _cache = cache;
17
18         public bool Add(IList<TLink> sequence)
19         {
20             var indexed = true;
21             var i = sequence.Count;
22             while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
23                 ⇨ { }
24         }
25     }
26 }

```

```

21         for (; i >= 1; i--)
22         {
23             _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
24         }
25         return indexed;
26     }
27
28     private bool IsIndexedWithIncrement(TLink source, TLink target)
29     {
30         var frequency = _cache.GetFrequency(source, target);
31         if (frequency == null)
32         {
33             return false;
34         }
35         var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
36         if (indexed)
37         {
38             _cache.IncrementFrequency(source, target);
39         }
40         return indexed;
41     }
42
43     public bool MightContain(ICollection<TLink> sequence)
44     {
45         var indexed = true;
46         var i = sequence.Count;
47         while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
48         return indexed;
49     }
50
51     private bool IsIndexed(TLink source, TLink target)
52     {
53         var frequency = _cache.GetFrequency(source, target);
54         if (frequency == null)
55         {
56             return false;
57         }
58         return !_equalityComparer.Equals(frequency.Frequency, default);
59     }
60 }
61 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Incremeters;
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 FrequencyIncrementingSequenceIndex<TLink> : SequenceIndex<TLink>,
10         ↳ ISequenceIndex<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↳ EqualityComparer<TLink>.Default;
14
15         private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
16         private readonly IIncrementer<TLink> _frequencyIncrementer;
17
18         public FrequencyIncrementingSequenceIndex(ICollection<TLink> links, IProperty<TLink, TLink>
19             ↳ frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
20             : base(links)
21         {
22             _frequencyPropertyOperator = frequencyPropertyOperator;
23             _frequencyIncrementer = frequencyIncrementer;
24         }
25
26         public override bool Add(ICollection<TLink> sequence)
27         {
28             var indexed = true;
29             var i = sequence.Count;
30             while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
31                 ↳ { }
32             for (; i >= 1; i--)
33             {
34                 Increment(links.GetOrCreate(sequence[i - 1], sequence[i]));
35             }
36             return indexed;
37         }
38     }
39 }

```

```

34
35     private bool IsIndexedWithIncrement(TLink source, TLink target)
36     {
37         var link = Links.SearchOrDefault(source, target);
38         var indexed = !_equalityComparer.Equals(link, default);
39         if (indexed)
40         {
41             Increment(link);
42         }
43         return indexed;
44     }
45
46     private void Increment(TLink link)
47     {
48         var previousFrequency = _frequencyPropertyOperator.Get(link);
49         var frequency = _frequencyIncrementer.Increment(previousFrequency);
50         _frequencyPropertyOperator.Set(link, frequency);
51     }
52 }
53 }

```

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

```

1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Indexes
6  {
7      public interface ISequenceIndex<TLink>
8      {
9          /// <summary>
10         /// Индексирует последовательность глобально, и возвращает значение,
11         /// определяющие была ли запрошенная последовательность проиндексирована ранее.
12         /// </summary>
13         /// <param name="sequence">Последовательность для индексации.</param>
14         bool Add(IList<TLink> sequence);
15
16         bool MightContain(IList<TLink> sequence);
17     }
18 }

```

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

```

1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Indexes
6  {
7      public class SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↪ EqualityComparer<TLink>.Default;
11
12         public SequenceIndex(ILinks<TLink> links) : base(links) { }
13
14         public virtual bool Add(IList<TLink> sequence)
15         {
16             var indexed = true;
17             var i = sequence.Count;
18             while (--i >= 1 && (indexed =
19                 ↪ !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
20                 ↪ default))) { }
21             for (; i >= 1; i--)
22             {
23                 Links.GetOrCreate(sequence[i - 1], sequence[i]);
24             }
25             return indexed;
26         }
27
28         public virtual bool MightContain(IList<TLink> sequence)
29         {
30             var indexed = true;
31             var i = sequence.Count;
32             while (--i >= 1 && (indexed =
33                 ↪ !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
34                 ↪ default))) { }
35             return indexed;
36         }
37     }
38 }

```

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

```

1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Indexes
6  {
7      public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↳ EqualityComparer<TLink>.Default;
11
12          private readonly ISynchronizedLinks<TLink> _links;
13
14          public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;
15
16          public bool Add(IList<TLink> sequence)
17          {
18              var indexed = true;
19              var i = sequence.Count;
20              var links = _links.Unsync;
21              _links.SyncRoot.ExecuteReadOperation(() =>
22              {
23                  while (--i >= 1 && (indexed =
24                      ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
25                      ↳ sequence[i]), default))) { }
26              });
27              if (!indexed)
28              {
29                  _links.SyncRoot.ExecuteWriteOperation(() =>
30                  {
31                      for (; i >= 1; i--)
32                      {
33                          links.GetOrCreate(sequence[i - 1], sequence[i]);
34                      }
35                  });
36              }
37              return indexed;
38          }
39
40          public bool MightContain(IList<TLink> sequence)
41          {
42              var links = _links.Unsync;
43              return _links.SyncRoot.ExecuteReadOperation(() =>
44              {
45                  var indexed = true;
46                  var i = sequence.Count;
47                  while (--i >= 1 && (indexed =
48                      ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
49                      ↳ sequence[i]), default))) { }
50                  return indexed;
51              });
52          }
53      }
54  }

```

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

```

1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Indexes
6  {
7      public class Unindex<TLink> : ISequenceIndex<TLink>
8      {
9          public virtual bool Add(IList<TLink> sequence) => false;
10
11          public virtual bool MightContain(IList<TLink> sequence) => true;
12      }
13  }

```

1.84 ./Platform.Data.Doublets/Sequences/ListFiller.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
7  {
8      public class ListFiller<TElement, TReturnConstant>

```

```

9 {
10     protected readonly List<TElement> _list;
11     protected readonly TReturnConstant _returnConstant;
12
13     public ListFiller(List<TElement> list, TReturnConstant returnConstant)
14     {
15         _list = list;
16         _returnConstant = returnConstant;
17     }
18
19     public ListFiller(List<TElement> list) : this(list, default) { }
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     public void Add(TElement element) => _list.Add(element);
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     public bool AddAndReturnTrue(TElement element)
26     {
27         _list.Add(element);
28         return true;
29     }
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     public bool AddFirstAndReturnTrue(ICollection<TElement> collection)
33     {
34         _list.Add(collection[0]);
35         return true;
36     }
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     public TReturnConstant AddAndReturnConstant(TElement element)
40     {
41         _list.Add(element);
42         return _returnConstant;
43     }
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> collection)
47     {
48         _list.Add(collection[0]);
49         return _returnConstant;
50     }
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     public TReturnConstant AddAllValuesAndReturnConstant(ICollection<TElement> collection)
54     {
55         for (int i = 1; i < collection.Count; i++)
56         {
57             _list.Add(collection[i]);
58         }
59         return _returnConstant;
60     }
61 }
62 }

```

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

```

1 using System;
2 using LinkIndex = System.UInt64;
3 using System.Collections.Generic;
4 using Stack = System.Collections.Generic.Stack<ulong>;
5 using System.Linq;
6 using System.Text;
7 using Platform.Collections;
8 using Platform.Collections.Sets;
9 using Platform.Collections.Stacks;
10 using Platform.Data.Exceptions;
11 using Platform.Data.Sequences;
12 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
13 using Platform.Data.Doublets.Sequences.Walkers;
14
15 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
16
17 namespace Platform.Data.Doublets.Sequences
18 {
19     partial class Sequences
20     {
21         #region Create All Variants (Not Practical)
22
23         /// <remarks>
24         /// Number of links that is needed to generate all variants for
25         /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.

```

```

26     /// </remarks>
27     public ulong[] CreateAllVariants2(ulong[] sequence)
28     {
29         return _sync.ExecuteWriteOperation(() =>
30         {
31             if (sequence.IsNullOrEmpty())
32             {
33                 return new ulong[0];
34             }
35             Links.EnsureLinkExists(sequence);
36             if (sequence.Length == 1)
37             {
38                 return sequence;
39             }
40             return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
41         });
42     }
43
44     private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
45     {
46         #if DEBUG
47             if ((stopAt - startAt) < 0)
48             {
49                 throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
50                     ↪ меньше или равен stopAt");
51             }
52             #endif
53             if ((stopAt - startAt) == 0)
54             {
55                 return new[] { sequence[startAt] };
56             }
57             if ((stopAt - startAt) == 1)
58             {
59                 return new[] { Links.Unsync.CreateAndUpdate(sequence[startAt], sequence[stopAt])
60                     ↪ };
61             }
62             var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
63             var last = 0;
64             for (var splitter = startAt; splitter < stopAt; splitter++)
65             {
66                 var left = CreateAllVariants2Core(sequence, startAt, splitter);
67                 var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
68                 for (var i = 0; i < left.Length; i++)
69                 {
70                     for (var j = 0; j < right.Length; j++)
71                     {
72                         var variant = Links.Unsync.CreateAndUpdate(left[i], right[j]);
73                         if (variant == Constants.Null)
74                         {
75                             throw new NotImplementedException("Creation cancellation is not
76                                 ↪ implemented.");
77                         }
78                         variants[last++] = variant;
79                     }
80                 }
81             }
82             return variants;
83         }
84
85         public List<ulong> CreateAllVariants1(params ulong[] sequence)
86         {
87             return _sync.ExecuteWriteOperation(() =>
88             {
89                 if (sequence.IsNullOrEmpty())
90                 {
91                     return new List<ulong>();
92                 }
93                 Links.Unsync.EnsureLinkExists(sequence);
94                 if (sequence.Length == 1)
95                 {
96                     return new List<ulong> { sequence[0] };
97                 }
98                 var results = new
99                     ↪ List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
100                 return CreateAllVariants1Core(sequence, results);
101             });
102         }

```

```

100 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
101 {
102     if (sequence.Length == 2)
103     {
104         var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);
105         if (link == Constants.Null)
106         {
107             throw new NotImplementedException("Creation cancellation is not
108                 ↪ implemented.");
109         }
110         results.Add(link);
111         return results;
112     }
113     var innerSequenceLength = sequence.Length - 1;
114     var innerSequence = new ulong[innerSequenceLength];
115     for (var li = 0; li < innerSequenceLength; li++)
116     {
117         var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
118         if (link == Constants.Null)
119         {
120             throw new NotImplementedException("Creation cancellation is not
121                 ↪ implemented.");
122         }
123         for (var isi = 0; isi < li; isi++)
124         {
125             innerSequence[isi] = sequence[isi];
126         }
127         innerSequence[li] = link;
128         for (var isi = li + 1; isi < innerSequenceLength; isi++)
129         {
130             innerSequence[isi] = sequence[isi + 1];
131         }
132         CreateAllVariants1Core(innerSequence, results);
133     }
134     return results;
135 }
136
137 #endregion
138
139 public HashSet<ulong> Each1(params ulong[] sequence)
140 {
141     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
142     Each1(link =>
143     {
144         if (!visitedLinks.Contains(link))
145         {
146             visitedLinks.Add(link); // изучить почему случаются повторы
147         }
148         return true;
149     }, sequence);
150     return visitedLinks;
151 }
152
153 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
154 {
155     if (sequence.Length == 2)
156     {
157         Links.Unsync.Each(sequence[0], sequence[1], handler);
158     }
159     else
160     {
161         var innerSequenceLength = sequence.Length - 1;
162         for (var li = 0; li < innerSequenceLength; li++)
163         {
164             var left = sequence[li];
165             var right = sequence[li + 1];
166             if (left == 0 && right == 0)
167             {
168                 continue;
169             }
170             var linkIndex = li;
171             ulong[] innerSequence = null;
172             Links.Unsync.Each(doublet =>
173             {
174                 if (innerSequence == null)
175                 {
176                     innerSequence = new ulong[innerSequenceLength];
177                     for (var isi = 0; isi < linkIndex; isi++)

```



```

176         {
177             innerSequence[isi] = sequence[isi];
178         }
179         for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
180         {
181             innerSequence[isi] = sequence[isi + 1];
182         }
183     }
184     innerSequence[linkIndex] = doublet[Constants.IndexPart];
185     Each1(handler, innerSequence);
186     return Constants.Continue;
187 }, Constants.Any, left, right);
188 }
189 }
190 }
191
192 public HashSet<ulong> EachPart(params ulong[] sequence)
193 {
194     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
195     EachPartCore(link =>
196     {
197         var linkIndex = link[Constants.IndexPart];
198         if (!visitedLinks.Contains(linkIndex))
199         {
200             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
201         }
202         return Constants.Continue;
203     }, sequence);
204     return visitedLinks;
205 }
206
207 public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
208 {
209     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
210     EachPartCore(link =>
211     {
212         var linkIndex = link[Constants.IndexPart];
213         if (!visitedLinks.Contains(linkIndex))
214         {
215             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
216             return handler(new LinkAddress<LinkIndex>(linkIndex));
217         }
218         return Constants.Continue;
219     }, sequence);
220 }
221
222 private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
223 ↪ sequence)
224 {
225     if (sequence.IsNullOrEmpty())
226     {
227         return;
228     }
229     Links.EnsureLinkIsAnyOrExists(sequence);
230     if (sequence.Length == 1)
231     {
232         var link = sequence[0];
233         if (link > 0)
234         {
235             handler(new LinkAddress<LinkIndex>(link));
236         }
237         else
238         {
239             Links.Each(Constants.Any, Constants.Any, handler);
240         }
241     }
242     else if (sequence.Length == 2)
243     {
244         //_links.Each(sequence[0], sequence[1], handler);
245         //  o_|      x_o ...
246         // x_|      |___|
247         Links.Each(sequence[1], Constants.Any, doublet =>
248         {
249             var match = Links.SearchOrDefault(sequence[0], doublet);
250             if (match != Constants.Null)
251             {
252                 handler(new LinkAddress<LinkIndex>(match));
253             }
254         });
255     }
256 }

```

```

253         return true;
254     });
255     // |_x      ... x_o
256     // |_o      |___|
257     Links.Unsync.Each(Constants.Any, sequence[0], doublet =>
258     {
259         var match = Links.SearchOrDefault(doublet, sequence[1]);
260         if (match != 0)
261         {
262             handler(new LinkAddress<LinkIndex>(match));
263         }
264         return true;
265     });
266     //      .x o_
267     //      |___|
268     PartialStepRight(x => handler(x), sequence[0], sequence[1]);
269 }
270 else
271 {
272     throw new NotImplementedException();
273 }
274 }
275
276 private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
277 {
278     Links.Unsync.Each(Constants.Any, left, doublet =>
279     {
280         StepRight(handler, doublet, right);
281         if (left != doublet)
282         {
283             PartialStepRight(handler, doublet, right);
284         }
285         return true;
286     });
287 }
288
289 private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
290 {
291     Links.Unsync.Each(left, Constants.Any, rightStep =>
292     {
293         TryStepRightUp(handler, right, rightStep);
294         return true;
295     });
296 }
297
298 private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
299 ↪ stepFrom)
300 {
301     var upStep = stepFrom;
302     var firstSource = Links.Unsync.GetTarget(upStep);
303     while (firstSource != right && firstSource != upStep)
304     {
305         upStep = firstSource;
306         firstSource = Links.Unsync.GetSource(upStep);
307     }
308     if (firstSource == right)
309     {
310         handler(new LinkAddress<LinkIndex>(stepFrom));
311     }
312 }
313
314 // TODO: Test
315 private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
316 {
317     Links.Unsync.Each(right, Constants.Any, doublet =>
318     {
319         StepLeft(handler, left, doublet);
320         if (right != doublet)
321         {
322             PartialStepLeft(handler, left, doublet);
323         }
324         return true;
325     });
326 }
327
328 private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
329 {
330     Links.Unsync.Each(Constants.Any, right, leftStep =>
331     {

```

```

331         TryStepLeftUp(handler, left, leftStep);
332         return true;
333     });
334 }
335
336 private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
337 {
338     var upStep = stepFrom;
339     var firstTarget = Links.Unsync.GetSource(upStep);
340     while (firstTarget != left && firstTarget != upStep)
341     {
342         upStep = firstTarget;
343         firstTarget = Links.Unsync.GetTarget(upStep);
344     }
345     if (firstTarget == left)
346     {
347         handler(new LinkAddress<LinkIndex>(stepFrom));
348     }
349 }
350
351 private bool StartsWith(ulong sequence, ulong link)
352 {
353     var upStep = sequence;
354     var firstSource = Links.Unsync.GetSource(upStep);
355     while (firstSource != link && firstSource != upStep)
356     {
357         upStep = firstSource;
358         firstSource = Links.Unsync.GetSource(upStep);
359     }
360     return firstSource == link;
361 }
362
363 private bool EndsWith(ulong sequence, ulong link)
364 {
365     var upStep = sequence;
366     var lastTarget = Links.Unsync.GetTarget(upStep);
367     while (lastTarget != link && lastTarget != upStep)
368     {
369         upStep = lastTarget;
370         lastTarget = Links.Unsync.GetTarget(upStep);
371     }
372     return lastTarget == link;
373 }
374
375 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
376 {
377     return _sync.ExecuteReadOperation(() =>
378     {
379         var results = new List<ulong>();
380         if (sequence.Length > 0)
381         {
382             Links.EnsureLinkExists(sequence);
383             var firstElement = sequence[0];
384             if (sequence.Length == 1)
385             {
386                 results.Add(firstElement);
387                 return results;
388             }
389             if (sequence.Length == 2)
390             {
391                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
392                 if (doublet != Constants.Null)
393                 {
394                     results.Add(doublet);
395                 }
396                 return results;
397             }
398             var linksInSequence = new HashSet<ulong>(sequence);
399             void handler(IList<LinkIndex> result)
400             {
401                 var resultIndex = result[Links.Constants.IndexPart];
402                 var filterPosition = 0;
403                 StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
404                     ↪ Links.Unsync.GetTarget,
405                     ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
406                     ↪ x =>
407                 {
408                     if (filterPosition == sequence.Length)
409                     {

```

```

408         filterPosition = -2; // Длиннее чем нужно
409         return false;
410     }
411     if (x != sequence[filterPosition])
412     {
413         filterPosition = -1;
414         return false; // Начинается иначе
415     }
416     filterPosition++;
417
418     return true;
419     });
420     if (filterPosition == sequence.Length)
421     {
422         results.Add(resultIndex);
423     }
424 }
425 if (sequence.Length >= 2)
426 {
427     StepRight(handler, sequence[0], sequence[1]);
428 }
429 var last = sequence.Length - 2;
430 for (var i = 1; i < last; i++)
431 {
432     PartialStepRight(handler, sequence[i], sequence[i + 1]);
433 }
434 if (sequence.Length >= 3)
435 {
436     StepLeft(handler, sequence[sequence.Length - 2],
437         ↪ sequence[sequence.Length - 1]);
438 }
439 }
440 return results;
441 });
442 }
443 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
444 {
445     return _sync.ExecuteReadOperation(() =>
446     {
447         var results = new HashSet<ulong>();
448         if (sequence.Length > 0)
449         {
450             Links.EnsureLinkExists(sequence);
451             var firstElement = sequence[0];
452             if (sequence.Length == 1)
453             {
454                 results.Add(firstElement);
455                 return results;
456             }
457             if (sequence.Length == 2)
458             {
459                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
460                 if (doublet != Constants.Null)
461                 {
462                     results.Add(doublet);
463                 }
464                 return results;
465             }
466             var matcher = new Matcher(this, sequence, results, null);
467             if (sequence.Length >= 2)
468             {
469                 StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
470             }
471             var last = sequence.Length - 2;
472             for (var i = 1; i < last; i++)
473             {
474                 PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
475                     ↪ sequence[i + 1]);
476             }
477             if (sequence.Length >= 3)
478             {
479                 StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
480                     ↪ sequence[sequence.Length - 1]);
481             }
482             return results;
483         }
484     });

```

```

483 }
484
485 public const int MaxSequenceFormatSize = 200;
486
487 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
488     => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
489
490 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
491     elementToString, bool insertComma, params LinkIndex[] knownElements) =>
492     Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
493         elementToString, insertComma, knownElements));
494
495 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
496     Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
497     LinkIndex[] knownElements)
498 {
499     var linksInSequence = new HashSet<ulong>(knownElements);
500     //var entered = new HashSet<ulong>();
501     var sb = new StringBuilder();
502     sb.Append('{');
503     if (links.Exists(sequenceLink))
504     {
505         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
506             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
507             entered.AddAndReturnVoid, x => { }, entered.DoNotContains
508             {
509                 if (insertComma && sb.Length > 1)
510                 {
511                     sb.Append(',');
512                 }
513                 //if (entered.Contains(element))
514                 //{
515                 //    sb.Append('{');
516                 //    elementToString(sb, element);
517                 //    sb.Append('}');
518                 //}
519                 //else
520                 elementToString(sb, element);
521                 if (sb.Length < MaxSequenceFormatSize)
522                 {
523                     return true;
524                 }
525                 sb.Append(insertComma ? ", ..." : "...");
526                 return false;
527             });
528     }
529     sb.Append('}');
530     return sb.ToString();
531 }
532
533 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
534     knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
535     knownElements);
536
537 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
538     LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
539     Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
540     sequenceLink, elementToString, insertComma, knownElements));
541
542 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
543     Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
544     LinkIndex[] knownElements)
545 {
546     var linksInSequence = new HashSet<ulong>(knownElements);
547     var entered = new HashSet<ulong>();
548     var sb = new StringBuilder();
549     sb.Append('{');
550     if (links.Exists(sequenceLink))
551     {
552         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
553             x => linksInSequence.Contains(x) || links.IsFullPoint(x),
554             entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
555             {
556                 if (insertComma && sb.Length > 1)
557                 {
558                     sb.Append(',');
559                 }
560             });
561     }
562     sb.Append('}');
563     return sb.ToString();
564 }

```

```

545         if (entered.Contains(element))
546         {
547             sb.Append('{');
548             elementToString(sb, element);
549             sb.Append('}');
550         }
551         else
552         {
553             elementToString(sb, element);
554         }
555         if (sb.Length < MaxSequenceFormatSize)
556         {
557             return true;
558         }
559         sb.Append(insertComma ? ", ..." : "...");
560         return false;
561     });
562 }
563 sb.Append('}');
564 return sb.ToString();
565 }
566
567 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
568 {
569     return _sync.ExecuteReadOperation(() =>
570     {
571         if (sequence.Length > 0)
572         {
573             Links.EnsureLinkExists(sequence);
574             var results = new HashSet<ulong>();
575             for (var i = 0; i < sequence.Length; i++)
576             {
577                 AllUsagesCore(sequence[i], results);
578             }
579             var filteredResults = new List<ulong>();
580             var linksInSequence = new HashSet<ulong>(sequence);
581             foreach (var result in results)
582             {
583                 var filterPosition = -1;
584                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
585                     ↪ Links.Unsync.GetTarget,
586                     ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
587                     ↪ x =>
588                     {
589                         if (filterPosition == (sequence.Length - 1))
590                         {
591                             return false;
592                         }
593                         if (filterPosition >= 0)
594                         {
595                             if (x == sequence[filterPosition + 1])
596                             {
597                                 filterPosition++;
598                             }
599                             else
600                             {
601                                 return false;
602                             }
603                         }
604                         if (filterPosition < 0)
605                         {
606                             if (x == sequence[0])
607                             {
608                                 filterPosition = 0;
609                             }
610                         }
611                         return true;
612                     }
613                 );
614                 if (filterPosition == (sequence.Length - 1))
615                 {
616                     filteredResults.Add(result);
617                 }
618             }
619             return filteredResults;
620         }
621         return new List<ulong>();
622     });
623 }

```

```

622 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
623 {
624     return _sync.ExecuteReadOperation(() =>
625     {
626         if (sequence.Length > 0)
627         {
628             Links.EnsureLinkExists(sequence);
629             var results = new HashSet<ulong>();
630             for (var i = 0; i < sequence.Length; i++)
631             {
632                 AllUsagesCore(sequence[i], results);
633             }
634             var filteredResults = new HashSet<ulong>();
635             var matcher = new Matcher(this, sequence, filteredResults, null);
636             matcher.AddAllPartialMatchedToResults(results);
637             return filteredResults;
638         }
639         return new HashSet<ulong>();
640     });
641 }
642
643 public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
644 ↪ params ulong[] sequence)
645 {
646     return _sync.ExecuteReadOperation(() =>
647     {
648         if (sequence.Length > 0)
649         {
650             Links.EnsureLinkExists(sequence);
651
652             var results = new HashSet<ulong>();
653             var filteredResults = new HashSet<ulong>();
654             var matcher = new Matcher(this, sequence, filteredResults, handler);
655             for (var i = 0; i < sequence.Length; i++)
656             {
657                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
658                 {
659                     return false;
660                 }
661             }
662             return true;
663         }
664         return true;
665     });
666 }
667
668 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
669 //{
670 //    return Sync.ExecuteReadOperation(() =>
671 //    {
672 //        if (sequence.Length > 0)
673 //        {
674 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
675
676 //            var firstResults = new HashSet<ulong>();
677 //            var lastResults = new HashSet<ulong>();
678
679 //            var first = sequence.First(x => x != LinksConstants.Any);
680 //            var last = sequence.Last(x => x != LinksConstants.Any);
681
682 //            AllUsagesCore(first, firstResults);
683 //            AllUsagesCore(last, lastResults);
684
685 //            firstResults.IntersectWith(lastResults);
686
687 //            //for (var i = 0; i < sequence.Length; i++)
688 //            //    AllUsagesCore(sequence[i], results);
689
690 //            var filteredResults = new HashSet<ulong>();
691 //            var matcher = new Matcher(this, sequence, filteredResults, null);
692 //            matcher.AddAllPartialMatchedToResults(firstResults);
693 //            return filteredResults;
694 //        }
695
696 //        return new HashSet<ulong>();
697 //    });
698 //}
699 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)

```

```

700 {
701     return _sync.ExecuteReadOperation((Func<HashSet<ulong>>)(() =>
702     {
703         if (sequence.Length > 0)
704         {
705             ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links,
706                 ↳ (IList<ulong>)sequence);
707             var firstResults = new HashSet<ulong>();
708             var lastResults = new HashSet<ulong>();
709             var first = sequence.First(x => x != Constants.Any);
710             var last = sequence.Last(x => x != Constants.Any);
711             AllUsagesCore(first, firstResults);
712             AllUsagesCore(last, lastResults);
713             firstResults.IntersectWith(lastResults);
714             //for (var i = 0; i < sequence.Length; i++)
715             //    AllUsagesCore(sequence[i], results);
716             var filteredResults = new HashSet<ulong>();
717             var matcher = new Matcher(this, sequence, filteredResults, null);
718             matcher.AddAllPartialMatchedToResults(firstResults);
719             return filteredResults;
720         }
721         return new HashSet<ulong>();
722     }));
723 }
724 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
725     ↳ IList<ulong> sequence)
726 {
727     return _sync.ExecuteReadOperation(() =>
728     {
729         if (sequence.Count > 0)
730         {
731             Links.EnsureLinkExists(sequence);
732             var results = new HashSet<LinkIndex>();
733             //var nextResults = new HashSet<ulong>();
734             //for (var i = 0; i < sequence.Length; i++)
735             //{
736             //    AllUsagesCore(sequence[i], nextResults);
737             //    if (results.IsNullOrEmpty())
738             //    {
739             //        results = nextResults;
740             //        nextResults = new HashSet<ulong>();
741             //    }
742             //    else
743             //    {
744             //        results.IntersectWith(nextResults);
745             //        nextResults.Clear();
746             //    }
747             //}
748             var collector1 = new AllUsagesCollector1(Links.Unsync, results);
749             collector1.Collect(Links.Unsync.GetLink(sequence[0]));
750             var next = new HashSet<ulong>();
751             for (var i = 1; i < sequence.Count; i++)
752             {
753                 var collector = new AllUsagesCollector1(Links.Unsync, next);
754                 collector.Collect(Links.Unsync.GetLink(sequence[i]));
755                 results.IntersectWith(next);
756                 next.Clear();
757             }
758             var filteredResults = new HashSet<ulong>();
759             var matcher = new Matcher(this, sequence, filteredResults, null,
760                 ↳ readAsElements);
761             matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
762                 ↳ x)); // OrderBy is a Hack
763             return filteredResults;
764         }
765         return new HashSet<ulong>();
766     }));
767 }
768 // Does not work
769 //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
770 //    ↳ params ulong[] sequence)
771 //{
772 //    var visited = new HashSet<ulong>();
773 //    var results = new HashSet<ulong>();

```



```

//      var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
→ true; }, readAsElements);
//      var last = sequence.Length - 1;
//      for (var i = 0; i < last; i++)
//      {
//          PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
//      }
//      return results;
//}

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

```

```

844         return new List<ulong>();
845     }
846 }
847 //if (matches.Count > 0)
848 //{
849     var usages = new HashSet<ulong>();
850     for (int i = 0; i < sequence.Length; i++)
851     {
852         AllUsagesCore(sequence[i], usages);
853     }
854     //for (int i = 0; i < matches[0].Count; i++)
855     //    AllUsagesCore(matches[0][i], usages);
856     //usages.UnionWith(matches[0]);
857     return usages.ToList();
858 }
859 var firstLinkUsages = new HashSet<ulong>();
860 AllUsagesCore(sequence[0], firstLinkUsages);
861 firstLinkUsages.Add(sequence[0]);
862 //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
863 //    sequence[0] }; // or all sequences, containing this element?
864 //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
865 //    1).ToList();
866 var results = new HashSet<ulong>();
867 foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
868     firstLinkUsages, 1))
869 {
870     AllUsagesCore(match, results);
871 }
872 return results.ToList();
873 }
874 return new List<ulong>();
875 });
876 }
877
878 /// <remarks>
879 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
880 /// </remarks>
881 public HashSet<ulong> AllUsages(ulong link)
882 {
883     return _sync.ExecuteReadOperation(() =>
884     {
885         var usages = new HashSet<ulong>();
886         AllUsagesCore(link, usages);
887         return usages;
888     });
889 }
890
891 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
892 // той связи с которой начинался поиск (STTTSSSTT),
893 // причём достаточно одного бита для хранения перехода влево или вправо
894 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
895 {
896     bool handler(ulong doublet)
897     {
898         if (usages.Add(doublet))
899         {
900             AllUsagesCore(doublet, usages);
901         }
902         return true;
903     }
904     Links.Unsync.Each(link, Constants.Any, handler);
905     Links.Unsync.Each(Constants.Any, link, handler);
906 }
907
908 public HashSet<ulong> AllBottomUsages(ulong link)
909 {
910     return _sync.ExecuteReadOperation(() =>
911     {
912         var visits = new HashSet<ulong>();
913         var usages = new HashSet<ulong>();
914         AllBottomUsagesCore(link, visits, usages);
915         return usages;
916     });
917 }
918
919 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
920     usages)
921 {

```

```

917     bool handler(ulong doublet)
918     {
919         if (visits.Add(doublet))
920         {
921             AllBottomUsagesCore(doublet, visits, usages);
922         }
923         return true;
924     }
925     if (Links.Unsync.Count(Constants.Any, link) == 0)
926     {
927         usages.Add(link);
928     }
929     else
930     {
931         Links.Unsync.Each(link, Constants.Any, handler);
932         Links.Unsync.Each(Constants.Any, link, handler);
933     }
934 }
935
936 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
937 {
938     if (Options.UseSequenceMarker)
939     {
940         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
941             ↪ Options.MarkedSequenceMatcher, symbol);
942         return counter.Count();
943     }
944     else
945     {
946         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
947             ↪ symbol);
948         return counter.Count();
949     }
950 }
951
952 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
953     ↪ LinkIndex> outerHandler)
954 {
955     bool handler(ulong doublet)
956     {
957         if (usages.Add(doublet))
958         {
959             if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
960             {
961                 return false;
962             }
963             if (!AllUsagesCore1(doublet, usages, outerHandler))
964             {
965                 return false;
966             }
967         }
968         return true;
969     }
970     return Links.Unsync.Each(link, Constants.Any, handler)
971         && Links.Unsync.Each(Constants.Any, link, handler);
972 }
973
974 public void CalculateAllUsages(ulong[] totals)
975 {
976     var calculator = new AllUsagesCalculator(Links, totals);
977     calculator.Calculate();
978 }
979
980 public void CalculateAllUsages2(ulong[] totals)
981 {
982     var calculator = new AllUsagesCalculator2(Links, totals);
983     calculator.Calculate();
984 }
985
986 private class AllUsagesCalculator
987 {
988     private readonly SynchronizedLinks<ulong> _links;
989     private readonly ulong[] _totals;
990
991     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
992     {
993         _links = links;
994         _totals = totals;
995     }
996 }

```

```

993     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
994         ↪ CalculateCore);
995
996     private bool CalculateCore(ulong link)
997     {
998         if (_totals[link] == 0)
999         {
1000             var total = 1UL;
1001             _totals[link] = total;
1002             var visitedChildren = new HashSet<ulong>();
1003             bool linkCalculator(ulong child)
1004             {
1005                 if (link != child && visitedChildren.Add(child))
1006                 {
1007                     total += _totals[child] == 0 ? 1 : _totals[child];
1008                 }
1009                 return true;
1010             }
1011             _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1012             _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1013             _totals[link] = total;
1014         }
1015         return true;
1016     }
1017 }
1018
1019 private class AllUsagesCalculator2
1020 {
1021     private readonly SynchronizedLinks<ulong> _links;
1022     private readonly ulong[] _totals;
1023
1024     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1025     {
1026         _links = links;
1027         _totals = totals;
1028     }
1029
1030     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1031         ↪ CalculateCore);
1032
1033     private bool IsElement(ulong link)
1034     {
1035         // _linksInSequence.Contains(link) ||
1036         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1037             ↪ link;
1038     }
1039
1040     private bool CalculateCore(ulong link)
1041     {
1042         // TODO: Проработать защиту от заикливания
1043         // Основано на SequenceWalker.WalkLeft
1044         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1045         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1046         Func<ulong, bool> isElement = IsElement;
1047         void visitLeaf(ulong parent)
1048         {
1049             if (link != parent)
1050             {
1051                 _totals[parent]++;
1052             }
1053         }
1054         void visitNode(ulong parent)
1055         {
1056             if (link != parent)
1057             {
1058                 _totals[parent]++;
1059             }
1060         }
1061         var stack = new Stack();
1062         var element = link;
1063         if (isElement(element))
1064         {
1065             visitLeaf(element);
1066         }
1067         else
1068         {
1069             while (true)
1070             {

```

```

1069         if (isElement(element))
1070         {
1071             if (stack.Count == 0)
1072             {
1073                 break;
1074             }
1075             element = stack.Pop();
1076             var source = getSource(element);
1077             var target = getTarget(element);
1078             // Обработка элемента
1079             if (isElement(target))
1080             {
1081                 visitLeaf(target);
1082             }
1083             if (isElement(source))
1084             {
1085                 visitLeaf(source);
1086             }
1087             element = source;
1088         }
1089         else
1090         {
1091             stack.Push(element);
1092             visitNode(element);
1093             element = getTarget(element);
1094         }
1095     }
1096 }
1097 _totals[link]++;
1098 return true;
1099 }
1100 }
1101
1102 private class AllUsagesCollector
1103 {
1104     private readonly ILinks<ulong> _links;
1105     private readonly HashSet<ulong> _usages;
1106
1107     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1108     {
1109         _links = links;
1110         _usages = usages;
1111     }
1112
1113     public bool Collect(ulong link)
1114     {
1115         if (_usages.Add(link))
1116         {
1117             _links.Each(link, _links.Constants.Any, Collect);
1118             _links.Each(_links.Constants.Any, link, Collect);
1119         }
1120         return true;
1121     }
1122 }
1123
1124 private class AllUsagesCollector1
1125 {
1126     private readonly ILinks<ulong> _links;
1127     private readonly HashSet<ulong> _usages;
1128     private readonly ulong _continue;
1129
1130     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1131     {
1132         _links = links;
1133         _usages = usages;
1134         _continue = _links.Constants.Continue;
1135     }
1136
1137     public ulong Collect(ICollection<ulong> link)
1138     {
1139         var linkIndex = _links.GetIndex(link);
1140         if (_usages.Add(linkIndex))
1141         {
1142             _links.Each(Collect, _links.Constants.Any, linkIndex);
1143         }
1144         return _continue;
1145     }
1146 }
1147
1148 private class AllUsagesCollector2

```

```

1149 {
1150     private readonly ILinks<ulong> _links;
1151     private readonly BitString _usages;
1152
1153     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1154     {
1155         _links = links;
1156         _usages = usages;
1157     }
1158
1159     public bool Collect(ulong link)
1160     {
1161         if (_usages.Add((long)link))
1162         {
1163             _links.Each(link, _links.Constants.Any, Collect);
1164             _links.Each(_links.Constants.Any, link, Collect);
1165         }
1166         return true;
1167     }
1168 }
1169
1170 private class AllUsagesIntersectingCollector
1171 {
1172     private readonly SynchronizedLinks<ulong> _links;
1173     private readonly HashSet<ulong> _intersectWith;
1174     private readonly HashSet<ulong> _usages;
1175     private readonly HashSet<ulong> _enter;
1176
1177     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
↪ intersectWith, HashSet<ulong> usages)
1178     {
1179         _links = links;
1180         _intersectWith = intersectWith;
1181         _usages = usages;
1182         _enter = new HashSet<ulong>(); // защита от зацикливания
1183     }
1184
1185     public bool Collect(ulong link)
1186     {
1187         if (_enter.Add(link))
1188         {
1189             if (_intersectWith.Contains(link))
1190             {
1191                 _usages.Add(link);
1192             }
1193             _links.Unsync.Each(link, _links.Constants.Any, Collect);
1194             _links.Unsync.Each(_links.Constants.Any, link, Collect);
1195         }
1196         return true;
1197     }
1198 }
1199
1200 private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
↪ right)
1201 {
1202     TryStepLeftUp(handler, left, right);
1203     TryStepRightUp(handler, right, left);
1204 }
1205
1206 private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
↪ right)
1207 {
1208     // Direct
1209     if (left == right)
1210     {
1211         handler(new LinkAddress<LinkIndex>(left));
1212     }
1213     var doublet = Links.Unsync.SearchOrDefault(left, right);
1214     if (doublet != Constants.Null)
1215     {
1216         handler(new LinkAddress<LinkIndex>(doublet));
1217     }
1218     // Inner
1219     CloseInnerConnections(handler, left, right);
1220     // Outer
1221     StepLeft(handler, left, right);
1222     StepRight(handler, left, right);
1223     PartialStepRight(handler, left, right);
1224     PartialStepLeft(handler, left, right);

```

```

1225 }
1226
1227 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
    ↳ HashSet<ulong> previousMatchings, long startAt)
1228 {
1229     if (startAt >= sequence.Length) // ?
1230     {
1231         return previousMatchings;
1232     }
1233     var secondLinkUsages = new HashSet<ulong>();
1234     AllUsagesCore(sequence[startAt], secondLinkUsages);
1235     secondLinkUsages.Add(sequence[startAt]);
1236     var matchings = new HashSet<ulong>();
1237     var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
1238     //for (var i = 0; i < previousMatchings.Count; i++)
1239     foreach (var secondLinkUsage in secondLinkUsages)
1240     {
1241         foreach (var previousMatching in previousMatchings)
1242         {
1243             //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
1244             ↳ secondLinkUsage);
1245             StepRight(filler.AddFirstAndReturnConstant, previousMatching,
1246             ↳ secondLinkUsage);
1247             TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
1248             ↳ previousMatching);
1249             //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1250             ↳ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1251             ↳ желаемым результатам.
1252             PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
1253             ↳ secondLinkUsage);
1254         }
1255     }
1256     if (matchings.Count == 0)
1257     {
1258         return matchings;
1259     }
1260     return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1261 }
1262
1263 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
    ↳ links, params ulong[] sequence)
1264 {
1265     if (sequence == null)
1266     {
1267         return;
1268     }
1269     for (var i = 0; i < sequence.Length; i++)
1270     {
1271         if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
1272         ↳ !links.Exists(sequence[i]))
1273         {
1274             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1275             ↳ $"patternSequence[{i}]");
1276         }
1277     }
1278 }
1279
1280 // Pattern Matching -> Key To Triggers
1281 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1282 {
1283     return _sync.ExecuteReadOperation(() =>
1284     {
1285         patternSequence = Simplify(patternSequence);
1286         if (patternSequence.Length > 0)
1287         {
1288             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1289             var uniqueSequenceElements = new HashSet<ulong>();
1290             for (var i = 0; i < patternSequence.Length; i++)
1291             {
1292                 if (patternSequence[i] != Constants.Any && patternSequence[i] !=
1293                 ↳ ZeroOrMany)
1294                 {
1295                     uniqueSequenceElements.Add(patternSequence[i]);
1296                 }
1297             }
1298             var results = new HashSet<ulong>();
1299             foreach (var uniqueSequenceElement in uniqueSequenceElements)

```

```

1291         {
1292             AllUsagesCore(uniqueSequenceElement, results);
1293         }
1294         var filteredResults = new HashSet<ulong>();
1295         var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1296         matcher.AddAllPatternMatchedToResults(results);
1297         return filteredResults;
1298     }
1299     return new HashSet<ulong>();
1300 });
1301 }
1302
1303 // Найти все возможные связи между указанным списком связей.
1304 // Находит связи между всеми указанными связями в любом порядке.
1305 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
1306 //        несколько раз в последовательности)
1307 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1308 {
1309     return _sync.ExecuteReadOperation(() =>
1310     {
1311         var results = new HashSet<ulong>();
1312         if (linksToConnect.Length > 0)
1313         {
1314             Links.EnsureLinkExists(linksToConnect);
1315             AllUsagesCore(linksToConnect[0], results);
1316             for (var i = 1; i < linksToConnect.Length; i++)
1317             {
1318                 var next = new HashSet<ulong>();
1319                 AllUsagesCore(linksToConnect[i], next);
1320                 results.IntersectWith(next);
1321             }
1322             return results;
1323         }
1324     });
1325 }
1326
1327 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1328 {
1329     return _sync.ExecuteReadOperation(() =>
1330     {
1331         var results = new HashSet<ulong>();
1332         if (linksToConnect.Length > 0)
1333         {
1334             Links.EnsureLinkExists(linksToConnect);
1335             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1336             collector1.Collect(linksToConnect[0]);
1337             var next = new HashSet<ulong>();
1338             for (var i = 1; i < linksToConnect.Length; i++)
1339             {
1340                 var collector = new AllUsagesCollector(Links.Unsync, next);
1341                 collector.Collect(linksToConnect[i]);
1342                 results.IntersectWith(next);
1343                 next.Clear();
1344             }
1345             return results;
1346         }
1347     });
1348 }
1349
1350 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1351 {
1352     return _sync.ExecuteReadOperation(() =>
1353     {
1354         var results = new HashSet<ulong>();
1355         if (linksToConnect.Length > 0)
1356         {
1357             Links.EnsureLinkExists(linksToConnect);
1358             var collector1 = new AllUsagesCollector(Links, results);
1359             collector1.Collect(linksToConnect[0]);
1360             //AllUsagesCore(linksToConnect[0], results);
1361             for (var i = 1; i < linksToConnect.Length; i++)
1362             {
1363                 var next = new HashSet<ulong>();
1364                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1365                 collector.Collect(linksToConnect[i]);
1366                 //AllUsagesCore(linksToConnect[i], next);
1367                 //results.IntersectWith(next);
1368                 results = next;

```



```

1368     }
1369     }
1370     return results;
1371 });
1372 }
1373
1374 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1375 {
1376     return _sync.ExecuteReadOperation(() =>
1377     {
1378         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1379         ↪ BitArray((int)_links.Total + 1);
1380         if (linksToConnect.Length > 0)
1381         {
1382             Links.EnsureLinkExists(linksToConnect);
1383             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1384             collector1.Collect(linksToConnect[0]);
1385             for (var i = 1; i < linksToConnect.Length; i++)
1386             {
1387                 var next = new BitString((long)Links.Unsync.Count() + 1); //new
1388                 ↪ BitArray((int)_links.Total + 1);
1389                 var collector = new AllUsagesCollector2(Links.Unsync, next);
1390                 collector.Collect(linksToConnect[i]);
1391                 results = results.And(next);
1392             }
1393             return results.GetSetUInt64Indices();
1394         }
1395     });
1396 }
1397
1398 private static ulong[] Simplify(ulong[] sequence)
1399 {
1400     // Считаем новый размер последовательности
1401     long newLength = 0;
1402     var zeroOrManyStepped = false;
1403     for (var i = 0; i < sequence.Length; i++)
1404     {
1405         if (sequence[i] == ZeroOrMany)
1406         {
1407             if (zeroOrManyStepped)
1408             {
1409                 continue;
1410             }
1411             zeroOrManyStepped = true;
1412         }
1413         else
1414         {
1415             //if (zeroOrManyStepped) Is it efficient?
1416             zeroOrManyStepped = false;
1417         }
1418         newLength++;
1419     }
1420     // Строим новую последовательность
1421     zeroOrManyStepped = false;
1422     var newSequence = new ulong[newLength];
1423     long j = 0;
1424     for (var i = 0; i < sequence.Length; i++)
1425     {
1426         //var current = zeroOrManyStepped;
1427         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1428         //if (current && zeroOrManyStepped)
1429         //    continue;
1430         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1431         //if (zeroOrManyStepped && newZeroOrManyStepped)
1432         //    continue;
1433         //zeroOrManyStepped = newZeroOrManyStepped;
1434         if (sequence[i] == ZeroOrMany)
1435         {
1436             if (zeroOrManyStepped)
1437             {
1438                 continue;
1439             }
1440             zeroOrManyStepped = true;
1441         }
1442         else
1443         {
1444             //if (zeroOrManyStepped) Is it efficient?
1445             zeroOrManyStepped = false;
1446         }
1447         newSequence[j++] = sequence[i];
1448     }
1449     return newSequence;
1450 }

```

```

1445         newSequence[j++] = sequence[i];
1446     }
1447     return newSequence;
1448 }
1449
1450 public static void TestSimplify()
1451 {
1452     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
        ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1453     var simplifiedSequence = Simplify(sequence);
1454 }
1455
1456 public List<ulong> GetSimilarSequences() => new List<ulong>();
1457
1458 public void Prediction()
1459 {
1460     //_links
1461     //_sequences
1462 }
1463
1464 #region From Triplets
1465
1466 //public static void DeleteSequence(Link sequence)
1467 //{
1468 //}
1469
1470 public List<ulong> CollectMatchingSequences(ulong[] links)
1471 {
1472     if (links.Length == 1)
1473     {
1474         throw new Exception("Подпоследовательности с одним элементом не
        ↪ поддерживаются.");
1475     }
1476     var leftBound = 0;
1477     var rightBound = links.Length - 1;
1478     var left = links[leftBound++];
1479     var right = links[rightBound--];
1480     var results = new List<ulong>();
1481     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1482     return results;
1483 }
1484
1485 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
    ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1486 {
1487     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1488     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1489     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1490     {
1491         var nextLeftLink = middleLinks[leftBound];
1492         var elements = GetRightElements(leftLink, nextLeftLink);
1493         if (leftBound <= rightBound)
1494         {
1495             for (var i = elements.Length - 1; i >= 0; i--)
1496             {
1497                 var element = elements[i];
1498                 if (element != 0)
1499                 {
1500                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
        ↪ rightLink, rightBound, ref results);
1501                 }
1502             }
1503         }
1504         else
1505         {
1506             for (var i = elements.Length - 1; i >= 0; i--)
1507             {
1508                 var element = elements[i];
1509                 if (element != 0)
1510                 {
1511                     results.Add(element);
1512                 }
1513             }
1514         }
1515     }
1516     else
1517     {
1518         var nextRightLink = middleLinks[rightBound];

```

```

1519     var elements = GetLeftElements(rightLink, nextRightLink);
1520     if (leftBound <= rightBound)
1521     {
1522         for (var i = elements.Length - 1; i >= 0; i--)
1523         {
1524             var element = elements[i];
1525             if (element != 0)
1526             {
1527                 CollectMatchingSequences(leftLink, leftBound, middleLinks,
1528                     ↪ elements[i], rightBound - 1, ref results);
1529             }
1530         }
1531     }
1532     else
1533     {
1534         for (var i = elements.Length - 1; i >= 0; i--)
1535         {
1536             var element = elements[i];
1537             if (element != 0)
1538             {
1539                 results.Add(element);
1540             }
1541         }
1542     }
1543 }
1544
1545 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1546 {
1547     var result = new ulong[5];
1548     TryStepRight(startLink, rightLink, result, 0);
1549     Links.Each(Constants.Any, startLink, couple =>
1550     {
1551         if (couple != startLink)
1552         {
1553             if (TryStepRight(couple, rightLink, result, 2))
1554             {
1555                 return false;
1556             }
1557         }
1558         return true;
1559     });
1560     if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1561     {
1562         result[4] = startLink;
1563     }
1564     return result;
1565 }
1566
1567 public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1568 {
1569     var added = 0;
1570     Links.Each(startLink, Constants.Any, couple =>
1571     {
1572         if (couple != startLink)
1573         {
1574             var coupleTarget = Links.GetTarget(couple);
1575             if (coupleTarget == rightLink)
1576             {
1577                 result[offset] = couple;
1578                 if (++added == 2)
1579                 {
1580                     return false;
1581                 }
1582             }
1583             else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
1584                 ↪ == Net.And &&
1585             {
1586                 result[offset + 1] = couple;
1587                 if (++added == 2)
1588                 {
1589                     return false;
1590                 }
1591             }
1592         }
1593     });
1594     return added > 0;

```

```

1595 }
1596
1597 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1598 {
1599     var result = new ulong[5];
1600     TryStepLeft(startLink, leftLink, result, 0);
1601     Links.Each(startLink, Constants.Any, couple =>
1602     {
1603         if (couple != startLink)
1604         {
1605             if (TryStepLeft(couple, leftLink, result, 2))
1606             {
1607                 return false;
1608             }
1609         }
1610         return true;
1611     });
1612     if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1613     {
1614         result[4] = leftLink;
1615     }
1616     return result;
1617 }
1618
1619 public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1620 {
1621     var added = 0;
1622     Links.Each(Constants.Any, startLink, couple =>
1623     {
1624         if (couple != startLink)
1625         {
1626             var coupleSource = Links.GetSource(couple);
1627             if (coupleSource == leftLink)
1628             {
1629                 result[offset] = couple;
1630                 if (++added == 2)
1631                 {
1632                     return false;
1633                 }
1634             }
1635             else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1636                 ↪ == Net.And &&
1637             {
1638                 result[offset + 1] = couple;
1639                 if (++added == 2)
1640                 {
1641                     return false;
1642                 }
1643             }
1644         }
1645         return true;
1646     });
1647     return added > 0;
1648 }
1649
1650 #endregion
1651
1652 #region Walkers
1653
1654 public class PatternMatcher : RightSequenceWalker<ulong>
1655 {
1656     private readonly Sequences _sequences;
1657     private readonly ulong[] _patternSequence;
1658     private readonly HashSet<LinkIndex> _linksInSequence;
1659     private readonly HashSet<LinkIndex> _results;
1660
1661     #region Pattern Match
1662
1663     enum PatternBlockType
1664     {
1665         Undefined,
1666         Gap,
1667         Elements
1668     }
1669
1670     struct PatternBlock
1671     {
1672         public PatternBlockType Type;
1673         public long Start;
1674         public long Stop;
1675     }
1676

```

```

1674     }
1675
1676     private readonly List<PatternBlock> _pattern;
1677     private int _patternPosition;
1678     private long _sequencePosition;
1679
1680     #endregion
1681
1682     public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1683         ↳ HashSet<LinkIndex> results)
1684         : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1685     {
1686         _sequences = sequences;
1687         _patternSequence = patternSequence;
1688         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1689             ↳ _sequences.Constants.Any && x != ZeroOrMany));
1690         _results = results;
1691         _pattern = CreateDetailedPattern();
1692     }
1693
1694     protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
1695         ↳ base.IsElement(link);
1696
1697     public bool PatternMatch(LinkIndex sequenceToMatch)
1698     {
1699         _patternPosition = 0;
1700         _sequencePosition = 0;
1701         foreach (var part in Walk(sequenceToMatch))
1702         {
1703             if (!PatternMatchCore(part))
1704             {
1705                 break;
1706             }
1707         }
1708         return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1709             ↳ - 1 && _pattern[_patternPosition].Start == 0);
1710     }
1711
1712     private List<PatternBlock> CreateDetailedPattern()
1713     {
1714         var pattern = new List<PatternBlock>();
1715         var patternBlock = new PatternBlock();
1716         for (var i = 0; i < _patternSequence.Length; i++)
1717         {
1718             if (patternBlock.Type == PatternBlockType.Undefined)
1719             {
1720                 if (_patternSequence[i] == _sequences.Constants.Any)
1721                 {
1722                     patternBlock.Type = PatternBlockType.Gap;
1723                     patternBlock.Start = 1;
1724                     patternBlock.Stop = 1;
1725                 }
1726                 else if (_patternSequence[i] == ZeroOrMany)
1727                 {
1728                     patternBlock.Type = PatternBlockType.Gap;
1729                     patternBlock.Start = 0;
1730                     patternBlock.Stop = long.MaxValue;
1731                 }
1732                 else
1733                 {
1734                     patternBlock.Type = PatternBlockType.Elements;
1735                     patternBlock.Start = i;
1736                     patternBlock.Stop = i;
1737                 }
1738             }
1739             else if (patternBlock.Type == PatternBlockType.Elements)
1740             {
1741                 if (_patternSequence[i] == _sequences.Constants.Any)
1742                 {
1743                     pattern.Add(patternBlock);
1744                     patternBlock = new PatternBlock
1745                     {
1746                         Type = PatternBlockType.Gap,
1747                         Start = 1,
1748                         Stop = 1
1749                     };
1750                 }
1751                 else if (_patternSequence[i] == ZeroOrMany)
1752                 {
1753                     pattern.Add(patternBlock);
1754                 }
1755             }
1756         }
1757     }

```

```

1750         patternBlock = new PatternBlock
1751         {
1752             Type = PatternBlockType.Gap,
1753             Start = 0,
1754             Stop = long.MaxValue
1755         };
1756     }
1757     else
1758     {
1759         patternBlock.Stop = i;
1760     }
1761 }
1762 else // patternBlock.Type == PatternBlockType.Gap
1763 {
1764     if (_patternSequence[i] == _sequences.Constants.Any)
1765     {
1766         patternBlock.Start++;
1767         if (patternBlock.Stop < patternBlock.Start)
1768         {
1769             patternBlock.Stop = patternBlock.Start;
1770         }
1771     }
1772     else if (_patternSequence[i] == ZeroOrMany)
1773     {
1774         patternBlock.Stop = long.MaxValue;
1775     }
1776     else
1777     {
1778         pattern.Add(patternBlock);
1779         patternBlock = new PatternBlock
1780         {
1781             Type = PatternBlockType.Elements,
1782             Start = i,
1783             Stop = i
1784         };
1785     }
1786 }
1787 }
1788 if (patternBlock.Type != PatternBlockType.Undefined)
1789 {
1790     pattern.Add(patternBlock);
1791 }
1792 return pattern;
1793 }
1794
1795 // match: search for regexp anywhere in text
1796 //int match(char* regexp, char* text)
1797 //{
1798 //    do
1799 //    {
1800 //        } while (*text++ != '\0');
1801 //    return 0;
1802 //}
1803
1804 // matchhere: search for regexp at beginning of text
1805 //int matchhere(char* regexp, char* text)
1806 //{
1807 //    if (regexp[0] == '\0')
1808 //        return 1;
1809 //    if (regexp[1] == '*')
1810 //        return matchstar(regexp[0], regexp + 2, text);
1811 //    if (regexp[0] == '$' && regexp[1] == '\0')
1812 //        return *text == '\0';
1813 //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1814 //        return matchhere(regexp + 1, text + 1);
1815 //    return 0;
1816 //}
1817
1818 // matchstar: search for c*regexp at beginning of text
1819 //int matchstar(int c, char* regexp, char* text)
1820 //{
1821 //    do
1822 //    {
1823 //        /* a * matches zero or more instances */
1824 //        if (matchhere(regexp, text))
1825 //            return 1;
1826 //    } while (*text != '\0' && (*text++ == c || c == '.'));
1827 //    return 0;
1828 //}

```

```

1829 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
    ↳ long maximumGap)
1830 //{
1831 //    mininumGap = 0;
1832 //    maximumGap = 0;
1833 //    element = 0;
1834 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1835 //    {
1836 //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1837 //            mininumGap++;
1838 //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1839 //            maximumGap = long.MaxValue;
1840 //        else
1841 //            break;
1842 //    }
1843
1844 //    if (maximumGap < mininumGap)
1845 //        maximumGap = mininumGap;
1846 //}
1847
1848 private bool PatternMatchCore(LinkIndex element)
1849 {
1850     if (_patternPosition >= _pattern.Count)
1851     {
1852         _patternPosition = -2;
1853         return false;
1854     }
1855     var currentPatternBlock = _pattern[_patternPosition];
1856     if (currentPatternBlock.Type == PatternBlockType.Gap)
1857     {
1858         //var currentMatchingBlockLength = (_sequencePosition -
            ↳ _lastMatchedBlockPosition);
1859         if (_sequencePosition < currentPatternBlock.Start)
1860         {
1861             _sequencePosition++;
1862             return true; // Двигаемся дальше
1863         }
1864         // Это последний блок
1865         if (_pattern.Count == _patternPosition + 1)
1866         {
1867             _patternPosition++;
1868             _sequencePosition = 0;
1869             return false; // Полное соответствие
1870         }
1871         else
1872         {
1873             if (_sequencePosition > currentPatternBlock.Stop)
1874             {
1875                 return false; // Соответствие невозможно
1876             }
1877             var nextPatternBlock = _pattern[_patternPosition + 1];
1878             if (_patternSequence[nextPatternBlock.Start] == element)
1879             {
1880                 if (nextPatternBlock.Start < nextPatternBlock.Stop)
1881                 {
1882                     _patternPosition++;
1883                     _sequencePosition = 1;
1884                 }
1885                 else
1886                 {
1887                     _patternPosition += 2;
1888                     _sequencePosition = 0;
1889                 }
1890             }
1891         }
1892     }
1893     else // currentPatternBlock.Type == PatternBlockType.Elements
1894     {
1895         var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1896         if (_patternSequence[patternElementPosition] != element)
1897         {
1898             return false; // Соответствие невозможно
1899         }
1900         if (patternElementPosition == currentPatternBlock.Stop)
1901         {
1902             _patternPosition++;
1903             _sequencePosition = 0;
1904         }
1905         else

```

```

1906         {
1907             _sequencePosition++;
1908         }
1909     }
1910     return true;
1911     //if (_patternSequence[_patternPosition] != element)
1912     //    return false;
1913     //else
1914     //{
1915     //    _sequencePosition++;
1916     //    _patternPosition++;
1917     //    return true;
1918     //}
1919     ///////
1920     //if (_filterPosition == _patternSequence.Length)
1921     //{
1922     //    _filterPosition = -2; // Длиннее чем нужно
1923     //    return false;
1924     //}
1925     //if (element != _patternSequence[_filterPosition])
1926     //{
1927     //    _filterPosition = -1;
1928     //    return false; // Начинается иначе
1929     //}
1930     // _filterPosition++;
1931     //if (_filterPosition == (_patternSequence.Length - 1))
1932     //    return false;
1933     //if (_filterPosition >= 0)
1934     //{
1935     //    if (element == _patternSequence[_filterPosition + 1])
1936     //        _filterPosition++;
1937     //    else
1938     //        return false;
1939     //}
1940     //if (_filterPosition < 0)
1941     //{
1942     //    if (element == _patternSequence[0])
1943     //        _filterPosition = 0;
1944     //}
1945 }
1946
1947 public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
1948 {
1949     foreach (var sequenceToMatch in sequencesToMatch)
1950     {
1951         if (PatternMatch(sequenceToMatch))
1952         {
1953             _results.Add(sequenceToMatch);
1954         }
1955     }
1956 }
1957
1958 #endregion
1959 }
1960
1961 }

```

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

```

1  using Platform.Collections;
2  using Platform.Collections.Lists;
3  using Platform.Collections.Stacks;
4  using Platform.Data.Doublets.Sequences.Walkers;
5  using Platform.Singletons;
6  using Platform.Threading.Synchronization;
7  using System;
8  using System.Collections.Generic;
9  using System.Linq;
10 using System.Runtime.CompilerServices;
11 using LinkIndex = System.UInt64;
12
13 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14
15 namespace Platform.Data.Doublets.Sequences
16 {
17     /// <summary>
18     /// Представляет коллекцию последовательностей связей.
19     /// </summary>
20     /// <remarks>
21     /// Обязательно реализовать атомарность каждого публичного метода.

```



```

22  ///
23  /// TODO:
24  ///
25  /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
26  /// через естественную группировку по unicode типам, все whitespace вместе, все символы
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.
34  ///
35  /// Рост последовательности слева и справа.
36  /// Поиск со звёздочкой.
37  /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
38  /// так же проблема может быть решена при реализации дистанционных триггеров.
39  /// Нужны ли уникальные указатели вообще?
40  /// Что если обращение к информации будет происходить через содержимое всегда?
41  ///
42  /// Писать тесты.
43  ///
44  ///
45  /// Можно убрать зависимость от конкретной реализации Links,
46  /// на зависимость от абстрактного элемента, который может быть представлен несколькими
47  /// способами.
48  ///
49  /// Можно ли как-то сделать один общий интерфейс
50  ///
51  /// Блокчейн и/или гит для распределённой записи транзакций.
52  ///
53  </remarks>
54  public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
55  {
56  // (после завершения реализации Sequences)
57  /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
58  /// связей.</summary>
59  public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
60
61  public SequencesOptions<LinkIndex> Options { get; }
62  public SynchronizedLinks<LinkIndex> Links { get; }
63  private readonly ISynchronization _sync;
64
65  public LinksConstants<LinkIndex> Constants { get; }
66
67  public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
68  {
69  Links = links;
70  _sync = links.SyncRoot;
71  Options = options;
72  Options.ValidateOptions();
73  Options.InitOptions(Links);
74  Constants = links.Constants;
75  }
76
77  public Sequences(SynchronizedLinks<LinkIndex> links)
78  : this(links, new SequencesOptions<LinkIndex>())
79  {
80  }
81
82  public bool IsSequence(LinkIndex sequence)
83  {
84  return _sync.ExecuteReadOperation(() =>
85  {
86  if (Options.UseSequenceMarker)
87  {
88  return Options.MarkedSequenceMatcher.IsMatched(sequence);
89  }
90  return !Links.Unsync.IsPartialPoint(sequence);
91  });
92  }
93
94  [MethodImpl(MethodImplOptions.AggressiveInlining)]
95  private LinkIndex GetSequenceByElements(LinkIndex sequence)
96  {
97  if (Options.UseSequenceMarker)
98  {

```

```

94         return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
95     }
96     return sequence;
97 }
98
99 private LinkIndex GetSequenceElements(LinkIndex sequence)
100 {
101     if (Options.UseSequenceMarker)
102     {
103         var linkContents = new Link<ulong>(Links.GetLink(sequence));
104         if (linkContents.Source == Options.SequenceMarkerLink)
105         {
106             return linkContents.Target;
107         }
108         if (linkContents.Target == Options.SequenceMarkerLink)
109         {
110             return linkContents.Source;
111         }
112     }
113     return sequence;
114 }
115
116 #region Count
117
118 public LinkIndex Count(ICollection<LinkIndex> restrictions)
119 {
120     if (restrictions.IsNullOrEmpty())
121     {
122         return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
123     }
124     if (restrictions.Count == 1) // Первая связь это адрес
125     {
126         var sequenceIndex = restrictions[0];
127         if (sequenceIndex == Constants.Null)
128         {
129             return 0;
130         }
131         if (sequenceIndex == Constants.Any)
132         {
133             return Count(null);
134         }
135         if (Options.UseSequenceMarker)
136         {
137             return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
138         }
139         return Links.Exists(sequenceIndex) ? 1UL : 0;
140     }
141     throw new NotImplementedException();
142 }
143
144 private LinkIndex CountUsages(params LinkIndex[] restrictions)
145 {
146     if (restrictions.Length == 0)
147     {
148         return 0;
149     }
150     if (restrictions.Length == 1) // Первая связь это адрес
151     {
152         if (restrictions[0] == Constants.Null)
153         {
154             return 0;
155         }
156         var any = Constants.Any;
157         if (Options.UseSequenceMarker)
158         {
159             var elementsLink = GetSequenceElements(restrictions[0]);
160             var sequenceLink = GetSequenceByElements(elementsLink);
161             if (sequenceLink != Constants.Null)
162             {
163                 return Links.Count(any, sequenceLink) + Links.Count(any, elementsLink) -
164                     ↪ 1;
165             }
166             return Links.Count(any, elementsLink);
167         }
168         return Links.Count(any, restrictions[0]);
169     }
170     throw new NotImplementedException();
171 }

```

```

172 #endregion
173
174 #region Create
175
176 public LinkIndex Create(IList<LinkIndex> restrictions)
177 {
178     return _sync.ExecuteWriteOperation(() =>
179     {
180         if (restrictions.IsNullOrEmpty())
181         {
182             return Constants.Null;
183         }
184         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
185         return CreateCore(restrictions);
186     });
187 }
188
189 private LinkIndex CreateCore(IList<LinkIndex> restrictions)
190 {
191     LinkIndex[] sequence = restrictions.ExtractValues();
192     if (Options.UseIndex)
193     {
194         Options.Index.Add(sequence);
195     }
196     var sequenceRoot = default(LinkIndex);
197     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
198     {
199         var matches = Each(restrictions);
200         if (matches.Count > 0)
201         {
202             sequenceRoot = matches[0];
203         }
204     }
205     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
206     {
207         return CompactCore(sequence);
208     }
209     if (sequenceRoot == default)
210     {
211         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
212     }
213     if (Options.UseSequenceMarker)
214     {
215         return Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
216     }
217     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
218 }
219
220 #endregion
221
222 #region Each
223
224 public List<LinkIndex> Each(IList<LinkIndex> sequence)
225 {
226     var results = new List<LinkIndex>();
227     var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
228     Each(filler.AddFirstAndReturnConstant, sequence);
229     return results;
230 }
231
232 public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
→ restrictions)
233 {
234     return _sync.ExecuteReadOperation(() =>
235     {
236         if (restrictions.IsNullOrEmpty())
237         {
238             return Constants.Continue;
239         }
240         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
241         if (restrictions.Count == 1)
242         {
243             var link = restrictions[0];
244             var any = Constants.Any;
245             if (link == any)
246             {
247                 if (Options.UseSequenceMarker)
248                 {

```

```

249         return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
250             ↪ Options.SequenceMarkerLink, any));
251     }
252     else
253     {
254         return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
255             ↪ any));
256     }
257     if (Options.UseSequenceMarker)
258     {
259         var sequenceLinkValues = Links.Unsync.GetLink(link);
260         if (sequenceLinkValues[Constants.SourcePart] ==
261             ↪ Options.SequenceMarkerLink)
262         {
263             link = sequenceLinkValues[Constants.TargetPart];
264         }
265         var sequence =
266             ↪ Options.Walker.Walk(link).ToArray().ConvertToRestrictionsValues();
267         sequence[0] = link;
268         return handler(sequence);
269     }
270     else if (restrictions.Count == 2)
271     {
272         throw new NotImplementedException();
273     }
274     else if (restrictions.Count == 3)
275     {
276         return Links.Unsync.Each(handler, restrictions);
277     }
278     else
279     {
280         var sequence = restrictions.ExtractValues();
281         if (Options.UseIndex && !Options.Index.MightContain(sequence))
282         {
283             return Constants.Break;
284         }
285         return EachCore(handler, sequence);
286     }
287 }
288
289 private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
290     ↪ values)
291 {
292     var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
293     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
294     ↪ Id.
295     Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
296     ↪ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
297     ↪ matcher.HandleFullMatched;
298     //if (sequence.Length >= 2)
299     if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
300     {
301         return Constants.Break;
302     }
303     var last = values.Count - 2;
304     for (var i = 1; i < last; i++)
305     {
306         if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
307             ↪ Constants.Continue)
308         {
309             return Constants.Break;
310         }
311     }
312     if (values.Count >= 3)
313     {
314         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
315             ↪ != Constants.Continue)
316         {
317             return Constants.Break;
318         }
319     }
320     return Constants.Continue;
321 }

```

```

316 private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ left, LinkIndex right)
317 {
318     return Links.Unsync.Each(doublet =>
319     {
320         var doubletIndex = doublet[Constants.IndexPart];
321         if (StepRight(handler, doubletIndex, right) != Constants.Continue)
322         {
323             return Constants.Break;
324         }
325         if (left != doubletIndex)
326         {
327             return PartialStepRight(handler, doubletIndex, right);
328         }
329         return Constants.Continue;
330     }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
331 }
332
333 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));
334
335 private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ right, LinkIndex stepFrom)
336 {
337     var upStep = stepFrom;
338     var firstSource = Links.Unsync.GetTarget(upStep);
339     while (firstSource != right && firstSource != upStep)
340     {
341         upStep = firstSource;
342         firstSource = Links.Unsync.GetSource(upStep);
343     }
344     if (firstSource == right)
345     {
346         return handler(new LinkAddress<LinkIndex>(stepFrom));
347     }
348     return Constants.Continue;
349 }
350
351 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));
352
353 private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ left, LinkIndex stepFrom)
354 {
355     var upStep = stepFrom;
356     var firstTarget = Links.Unsync.GetSource(upStep);
357     while (firstTarget != left && firstTarget != upStep)
358     {
359         upStep = firstTarget;
360         firstTarget = Links.Unsync.GetTarget(upStep);
361     }
362     if (firstTarget == left)
363     {
364         return handler(new LinkAddress<LinkIndex>(stepFrom));
365     }
366     return Constants.Continue;
367 }
368
369 #endregion
370
371 #region Update
372
373 public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
374 {
375     var sequence = restrictions.ExtractValues();
376     var newSequence = substitution.ExtractValues();
377
378     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
379     {
380         return Constants.Null;
381     }
382     if (sequence.IsNullOrEmpty())
383     {
384         return Create(substitution);
385     }

```

```

386         if (newSequence.IsNullOrEmpty())
387         {
388             Delete(restrictions);
389             return Constants.Null;
390         }
391         return _sync.ExecuteWriteOperation((Func<ulong>) (() =>
392         {
393             ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
394             Links.EnsureLinkExists(newSequence);
395             return UpdateCore(sequence, newSequence);
396         }));
397     }
398
399     private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence)
400     {
401         LinkIndex bestVariant;
402         if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
403             ↪ !sequence.EqualTo(newSequence))
404         {
405             bestVariant = CompactCore(newSequence);
406         }
407         else
408         {
409             bestVariant = CreateCore(newSequence);
410         }
411         // TODO: Check all options only ones before loop execution
412         // Возможно нужно две версии Each, возвращающий фактические последовательности и с
413         ↪ маркером,
414         // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
415         ↪ можно получить имея только фактические последовательности.
416         foreach (var variant in Each(sequence))
417         {
418             if (variant != bestVariant)
419             {
420                 UpdateOneCore(variant, bestVariant);
421             }
422         }
423         return bestVariant;
424     }
425
426     private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
427     {
428         if (Options.UseGarbageCollection)
429         {
430             var sequenceElements = GetSequenceElements(sequence);
431             var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
432             var sequenceLink = GetSequenceByElements(sequenceElements);
433             var newSequenceElements = GetSequenceElements(newSequence);
434             var newSequenceLink = GetSequenceByElements(newSequenceElements);
435             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
436             {
437                 if (sequenceLink != Constants.Null)
438                 {
439                     Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
440                 }
441                 Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
442             }
443             ClearGarbage(sequenceElementsContents.Source);
444             ClearGarbage(sequenceElementsContents.Target);
445         }
446         else
447         {
448             if (Options.UseSequenceMarker)
449             {
450                 var sequenceElements = GetSequenceElements(sequence);
451                 var sequenceLink = GetSequenceByElements(sequenceElements);
452                 var newSequenceElements = GetSequenceElements(newSequence);
453                 var newSequenceLink = GetSequenceByElements(newSequenceElements);
454                 if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
455                 {
456                     if (sequenceLink != Constants.Null)
457                     {
458                         Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
459                     }
460                     Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
461                 }
462             }
463             else
464             {
465                 // ...
466             }
467         }
468     }

```

```

461         {
462             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
463             {
464                 Links.Unsync.MergeAndDelete(sequence, newSequence);
465             }
466         }
467     }
468 }
469
470 #endregion
471
472 #region Delete
473
474 public void Delete(IList<LinkIndex> restrictions)
475 {
476     _sync.ExecuteWriteOperation(() =>
477     {
478         var sequence = restrictions.ExtractValues();
479         // TODO: Check all options only ones before loop execution
480         foreach (var linkToDelete in Each(sequence))
481         {
482             DeleteOneCore(linkToDelete);
483         }
484     });
485 }
486
487 private void DeleteOneCore(LinkIndex link)
488 {
489     if (Options.UseGarbageCollection)
490     {
491         var sequenceElements = GetSequenceElements(link);
492         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
493         var sequenceLink = GetSequenceByElements(sequenceElements);
494         if (Options.UseCascadeDelete || CountUsages(link) == 0)
495         {
496             if (sequenceLink != Constants.Null)
497             {
498                 Links.Unsync.Delete(sequenceLink);
499             }
500             Links.Unsync.Delete(link);
501         }
502         ClearGarbage(sequenceElementsContents.Source);
503         ClearGarbage(sequenceElementsContents.Target);
504     }
505     else
506     {
507         if (Options.UseSequenceMarker)
508         {
509             var sequenceElements = GetSequenceElements(link);
510             var sequenceLink = GetSequenceByElements(sequenceElements);
511             if (Options.UseCascadeDelete || CountUsages(link) == 0)
512             {
513                 if (sequenceLink != Constants.Null)
514                 {
515                     Links.Unsync.Delete(sequenceLink);
516                 }
517                 Links.Unsync.Delete(link);
518             }
519         }
520         else
521         {
522             if (Options.UseCascadeDelete || CountUsages(link) == 0)
523             {
524                 Links.Unsync.Delete(link);
525             }
526         }
527     }
528 }
529
530 #endregion
531
532 #region Compactification
533
534 public void CompactAll()
535 {
536     _sync.ExecuteWriteOperation(() =>
537     {
538         var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
539         for (int i = 0; i < sequences.Count; i++)

```

```

540         {
541             var sequence = this.ToList(sequences[i]);
542             Compact(sequence.ConvertToRestrictionsValues());
543         }
544     });
545 }
546
547 /// <remarks>
548 /// bestVariant можно выбирать по максимальному числу использований,
549 /// но балансированный позволяет гарантировать уникальность (если есть возможность,
550 /// гарантировать его использование в других местах).
551 ///
552 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
553 /// </remarks>
554 public LinkIndex Compact(ICollection<LinkIndex> sequence)
555 {
556     return _sync.ExecuteWriteOperation(() =>
557     {
558         if (sequence.IsNullOrEmpty())
559         {
560             return Constants.Null;
561         }
562         Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
563         return CompactCore(sequence);
564     });
565 }
566
567 [MethodImpl(MethodImplOptions.AggressiveInlining)]
568 private LinkIndex CompactCore(ICollection<LinkIndex> sequence) => UpdateCore(sequence,
569     ↪ sequence);
570
571 #endregion
572
573 #region Garbage Collection
574
575 /// <remarks>
576 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
577 ↪ определить извне или в унаследованном классе
578 /// </remarks>
579 [MethodImpl(MethodImplOptions.AggressiveInlining)]
580 private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
581     ↪ !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
582
583 private void ClearGarbage(LinkIndex link)
584 {
585     if (IsGarbage(link))
586     {
587         var contents = new Link<ulong>(Links.GetLink(link));
588         Links.Unsync.Delete(link);
589         ClearGarbage(contents.Source);
590         ClearGarbage(contents.Target);
591     }
592 }
593
594 #endregion
595
596 #region Walkers
597
598 public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
599 {
600     return _sync.ExecuteReadOperation(() =>
601     {
602         var links = Links.Unsync;
603         foreach (var part in Options.Walker.Walk(sequence))
604         {
605             if (!handler(part))
606             {
607                 return false;
608             }
609         }
610         return true;
611     });
612 }
613
614 public class Matcher : RightSequenceWalker<LinkIndex>
615 {
616     private readonly Sequences _sequences;
617     private readonly ICollection<LinkIndex> _patternSequence;
618     private readonly HashSet<LinkIndex> _linksInSequence;

```



```

616 private readonly HashSet<LinkIndex> _results;
617 private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
618 private readonly HashSet<LinkIndex> _readAsElements;
619 private int _filterPosition;
620
621 public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
622   ↳ HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
623   ↳ HashSet<LinkIndex> readAsElements = null)
624     : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
625 {
626     _sequences = sequences;
627     _patternSequence = patternSequence;
628     _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
629   ↳ Links.Constants.Any && x != ZeroOrMany));
630     _results = results;
631     _stopableHandler = stopableHandler;
632     _readAsElements = readAsElements;
633 }
634
635 protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
636   ↳ (_readAsElements != null && _readAsElements.Contains(link)) ||
637   ↳ _linksInSequence.Contains(link);
638
639 public bool FullMatch(LinkIndex sequenceToMatch)
640 {
641     _filterPosition = 0;
642     foreach (var part in Walk(sequenceToMatch))
643     {
644         if (!FullMatchCore(part))
645         {
646             break;
647         }
648     }
649     return _filterPosition == _patternSequence.Count;
650 }
651
652 private bool FullMatchCore(LinkIndex element)
653 {
654     if (_filterPosition == _patternSequence.Count)
655     {
656         _filterPosition = -2; // Длиннее чем нужно
657         return false;
658     }
659     if (_patternSequence[_filterPosition] != Links.Constants.Any
660   && element != _patternSequence[_filterPosition])
661     {
662         _filterPosition = -1;
663         return false; // Начинается/Продолжается иначе
664     }
665     _filterPosition++;
666     return true;
667 }
668
669 public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
670 {
671     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
672     if (FullMatch(sequenceToMatch))
673     {
674         _results.Add(sequenceToMatch);
675     }
676 }
677
678 public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
679 {
680     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
681     if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
682     {
683         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
684     }
685     return Links.Constants.Continue;
686 }
687
688 public LinkIndex HandleFullMatchedSequence(IList<LinkIndex> restrictions)
689 {
690     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
691     var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
692     if (sequence != Links.Constants.Null && FullMatch(sequenceToMatch) &&
693   ↳ _results.Add(sequenceToMatch))
694     {
695

```

```

689         return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
690     }
691     return Links.Constants.Continue;
692 }
693
694 /// <remarks>
695 /// TODO: Add support for LinksConstants.Any
696 /// </remarks>
697 public bool PartialMatch(LinkIndex sequenceToMatch)
698 {
699     _filterPosition = -1;
700     foreach (var part in Walk(sequenceToMatch))
701     {
702         if (!PartialMatchCore(part))
703         {
704             break;
705         }
706     }
707     return _filterPosition == _patternSequence.Count - 1;
708 }
709
710 private bool PartialMatchCore(LinkIndex element)
711 {
712     if (_filterPosition == (_patternSequence.Count - 1))
713     {
714         return false; // Нашлось
715     }
716     if (_filterPosition >= 0)
717     {
718         if (element == _patternSequence[_filterPosition + 1])
719         {
720             _filterPosition++;
721         }
722         else
723         {
724             _filterPosition = -1;
725         }
726     }
727     if (_filterPosition < 0)
728     {
729         if (element == _patternSequence[0])
730         {
731             _filterPosition = 0;
732         }
733     }
734     return true; // Ищем дальше
735 }
736
737 public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
738 {
739     if (PartialMatch(sequenceToMatch))
740     {
741         _results.Add(sequenceToMatch);
742     }
743 }
744
745 public LinkIndex HandlePartialMatched(ICollection<LinkIndex> restrictions)
746 {
747     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
748     if (PartialMatch(sequenceToMatch))
749     {
750         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
751     }
752     return Links.Constants.Continue;
753 }
754
755 public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)
756 {
757     foreach (var sequenceToMatch in sequencesToMatch)
758     {
759         if (PartialMatch(sequenceToMatch))
760         {
761             _results.Add(sequenceToMatch);
762         }
763     }
764 }
765
766 public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
    ⇨ sequencesToMatch)

```

```

767     {
768         foreach (var sequenceToMatch in sequencesToMatch)
769         {
770             if (PartialMatch(sequenceToMatch))
771             {
772                 _readAsElements.Add(sequenceToMatch);
773                 _results.Add(sequenceToMatch);
774             }
775         }
776     }
777 }
778 #endregion
779 }
780 }
781 }

```

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

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Collections.Stacks;
5  using Platform.Converters;
6  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
7  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
8  using Platform.Data.Doublets.Sequences.Converters;
9  using Platform.Data.Doublets.Sequences.Walkers;
10 using Platform.Data.Doublets.Sequences.Indexes;
11 using Platform.Data.Doublets.Sequences.CriterionMatchers;
12
13 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14
15 namespace Platform.Data.Doublets.Sequences
16 {
17     public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
            ↳ ILinks<TLink> must contain GetConstants function.
18     {
19         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
20
21         public TLink SequenceMarkerLink { get; set; }
22         public bool UseCascadeUpdate { get; set; }
23         public bool UseCascadeDelete { get; set; }
24         public bool UseIndex { get; set; } // TODO: Update Index on sequence update/delete.
25         public bool UseSequenceMarker { get; set; }
26         public bool UseCompression { get; set; }

```

```

27 public bool UseGarbageCollection { get; set; }
28 public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get; set; }
29 public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }
30
31 public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher { get; set; }
32 public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
33 public ISequenceIndex<TLink> Index { get; set; }
34 public ISequenceWalker<TLink> Walker { get; set; }
35 public bool ReadFullSequence { get; set; }
36
37 // TODO: Реализовать компактификацию при чтении
38 //public bool EnforceSingleSequenceVersionOnRead { get; set; }
39 //public bool UseRequestMarker { get; set; }
40 //public bool StoreRequestResults { get; set; }
41
42 public void InitOptions(ISynchronizedLinks<TLink> links)
43 {
44     if (UseSequenceMarker)
45     {
46         if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
47         {
48             SequenceMarkerLink = links.CreatePoint();
49         }
50         else
51         {
52             if (!links.Exists(SequenceMarkerLink))
53             {
54                 var link = links.CreatePoint();
55                 if (!_equalityComparer.Equals(link, SequenceMarkerLink))
56                 {
57                     throw new InvalidOperationException("Cannot recreate sequence marker
58                         ↪ link.");
59                 }
60             }
61             if (MarkedSequenceMatcher == null)
62             {
63                 MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
64                     ↪ SequenceMarkerLink);
65             }
66         }
67     }
68     var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
69     if (UseCompression)
70     {
71         if (LinksToSequenceConverter == null)
72         {
73             ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
74             if (UseSequenceMarker)
75             {
76                 totalSequenceSymbolFrequencyCounter = new
77                     ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
78                     ↪ MarkedSequenceMatcher);
79             }
80             else
81             {
82                 totalSequenceSymbolFrequencyCounter = new
83                     ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
84             }
85             var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
86                 ↪ totalSequenceSymbolFrequencyCounter);
87             var compressingConverter = new CompressingConverter<TLink>(links,
88                 ↪ balancedVariantConverter, doubletFrequenciesCache);
89             LinksToSequenceConverter = compressingConverter;
90         }
91     }
92     else
93     {
94         if (LinksToSequenceConverter == null)
95         {
96             LinksToSequenceConverter = balancedVariantConverter;
97         }
98     }
99     if (UseIndex && Index == null)
100     {
101         Index = new SequenceIndex<TLink>(links);
102     }
103     if (Walker == null)
104     {

```

```

98         Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
99     }
100 }
101
102 public void ValidateOptions()
103 {
104     if (UseGarbageCollection && !UseSequenceMarker)
105     {
106         throw new NotSupportedException("To use garbage collection UseSequenceMarker
107             ↪ option must be on.");
108     }
109 }
110 }

```

1.89 ./Platform.Data.Doublets/Sequences/SetFiller.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
7  {
8      public class SetFiller<TElement, TReturnConstant>
9      {
10         protected readonly ISet<TElement> _set;
11         protected readonly TReturnConstant _returnConstant;
12
13         public SetFiller(ISet<TElement> set, TReturnConstant returnConstant)
14         {
15             _set = set;
16             _returnConstant = returnConstant;
17         }
18
19         public SetFiller(ISet<TElement> set) : this(set, default) { }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public void Add(TElement element) => _set.Add(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public bool AddAndReturnTrue(TElement element)
26         {
27             _set.Add(element);
28             return true;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public bool AddFirstAndReturnTrue(ICollection<TElement> collection)
33         {
34             _set.Add(collection[0]);
35             return true;
36         }
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public TReturnConstant AddAndReturnConstant(TElement element)
40         {
41             _set.Add(element);
42             return _returnConstant;
43         }
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> collection)
47         {
48             _set.Add(collection[0]);
49             return _returnConstant;
50         }
51     }
52 }

```

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

```

1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Walkers
6  {
7      public interface ISequenceWalker<TLink>
8      {
9          IEnumerable<TLink> Walk(TLink sequence);

```

```

10     }
11 }

```

1.91 ./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         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
13             ↪ isElement) : base(links, stack, isElement) { }
14
15         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
16             ↪ links.IsPartialPoint) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override TLink GetNextElementAfterPop(TLink element) =>
20             ↪ Links.GetSource(element);
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetNextElementAfterPush(TLink element) =>
24             ↪ Links.GetTarget(element);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override IEnumerable<TLink> WalkContents(TLink element)
28         {
29             var parts = Links.GetLink(element);
30             var start = Links.Constants.IndexPart + 1;
31             for (var i = parts.Count - 1; i >= start; i--)
32             {
33                 var part = parts[i];
34                 if (IsElement(part))
35                 {
36                     yield return part;
37                 }
38             }
39         }
40     }
41 }

```

1.92 ./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 =
17             ↪ EqualityComparer<TLink>.Default;
18
19         private readonly Func<TLink, bool> _isElement;
20
21         public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
22             ↪ base(links) => _isElement = isElement;
23
24         public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
25             ↪ Links.IsPartialPoint;
26
27         public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
28
29         public TLink[] ToArray(TLink sequence)
30         {
31             var length = 1;
32             var array = new TLink[length];
33             array[0] = sequence;
34         }
35     }
36 }

```

```

31         if (!_isElement(sequence))
32         {
33             return array;
34         }
35         bool hasElements;
36         do
37         {
38             length *= 2;
39 #if USEARRAYPOOL
40             var nextArray = ArrayPool.Allocate<ulong>(length);
41 #else
42             var nextArray = new TLink[length];
43 #endif
44             hasElements = false;
45             for (var i = 0; i < array.Length; i++)
46             {
47                 var candidate = array[i];
48                 if (!_equalityComparer.Equals(array[i], default))
49                 {
50                     continue;
51                 }
52                 var doubletOffset = i * 2;
53                 if (!_isElement(candidate))
54                 {
55                     nextArray[doubletOffset] = candidate;
56                 }
57                 else
58                 {
59                     var link = Links.GetLink(candidate);
60                     var linkSource = Links.GetSource(link);
61                     var linkTarget = Links.GetTarget(link);
62                     nextArray[doubletOffset] = linkSource;
63                     nextArray[doubletOffset + 1] = linkTarget;
64                     if (!hasElements)
65                     {
66                         hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
67                     }
68                 }
69             }
70 #if USEARRAYPOOL
71             if (array.Length > 1)
72             {
73                 ArrayPool.Free(array);
74             }
75 #endif
76             array = nextArray;
77         }
78         while (hasElements);
79         var filledElementsCount = CountFilledElements(array);
80         if (filledElementsCount == array.Length)
81         {
82             return array;
83         }
84         else
85         {
86             return CopyFilledElements(array, filledElementsCount);
87         }
88     }
89
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
92     {
93         var finalArray = new TLink[filledElementsCount];
94         for (int i = 0, j = 0; i < array.Length; i++)
95         {
96             if (!_equalityComparer.Equals(array[i], default))
97             {
98                 finalArray[j] = array[i];
99                 j++;
100             }
101         }
102 #if USEARRAYPOOL
103         ArrayPool.Free(array);
104 #endif
105         return finalArray;
106     }
107
108     [MethodImpl(MethodImplOptions.AggressiveInlining)]
109     private static int CountFilledElements(TLink[] array)

```

```

110     {
111         var count = 0;
112         for (var i = 0; i < array.Length; i++)
113         {
114             if (!_equalityComparer.Equals(array[i], default))
115             {
116                 count++;
117             }
118         }
119         return count;
120     }
121 }
122 }

```

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

```

1.94 ./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         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
17             ↪ isElement) : base(links)
18         {
19             _stack = stack;
20             _isElement = isElement;
21         }
22     }
23 }

```



```

21     protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
    ↪     stack, links.IsPartialPoint)
22     {
23     }
24
25     public IEnumerable<TLink> Walk(TLink sequence)
26     {
27         _stack.Clear();
28         var element = sequence;
29         if (IsElement(element))
30         {
31             yield return element;
32         }
33         else
34         {
35             while (true)
36             {
37                 if (IsElement(element))
38                 {
39                     if (_stack.IsEmpty)
40                     {
41                         break;
42                     }
43                     element = _stack.Pop();
44                     foreach (var output in WalkContents(element))
45                     {
46                         yield return output;
47                     }
48                     element = GetNextElementAfterPop(element);
49                 }
50                 else
51                 {
52                     _stack.Push(element);
53                     element = GetNextElementAfterPush(element);
54                 }
55             }
56         }
57     }
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected abstract TLink GetNextElementAfterPop(TLink element);
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected abstract TLink GetNextElementAfterPush(TLink element);
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     protected abstract IEnumerable<TLink> WalkContents(TLink element);
70 }
71 }

```

1.95 ./Platform.Data.Doublets/Stacks/Stack.cs

```

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

```

```

26
27     public TLink Peek() => _links.GetTarget(GetTop());
28
29     public TLink Pop()
30     {
31         var element = Peek();
32         if (!_equalityComparer.Equals(element, _stack))
33         {
34             var top = GetTop();
35             var previousTop = _links.GetSource(top);
36             _links.Update(_stack, GetStackMarker(), previousTop);
37             _links.Delete(top);
38         }
39         return element;
40     }
41
42     public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
43     ↪ _links.GetOrCreate(GetTop(), element));
44 }

```

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

```

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

```

1.97 ./Platform.Data.Doublets/SynchronizedLinks.cs

```

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

```

```

35     public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
        ↳ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
        ↳ Unsync.Update);
36     public void Delete(IList<TLinkAddress> restrictions) =>
        ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
37
38     //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
        ↳ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
39     //{
40     //    if (restriction != null && substitution != null &&
        ↳ !substitution.EqualTo(restriction))
41     //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
        ↳ substitution, substitutedHandler, Unsync.Trigger);
42
43     //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
        ↳ substitutedHandler, Unsync.Trigger);
44     //}
45 }
46 }

```

1.98 ./Platform.Data.Doublets/UInt64LinksExtensions.cs

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using Platform.Singletons;
5  using Platform.Data.Exceptions;
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         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
17
18
19
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         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
            ↳ Func<Link<ulong>, bool> isElement, bool renderIndex = false, bool renderDebug =
            ↳ false)
38         {
39             var sb = new StringBuilder();
40             var visited = new HashSet<ulong>();
41             links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
                ↳ innerSb.Append(link.Index), renderIndex, renderDebug);
42             return sb.ToString();
43         }
44
45         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)
46         {
47             var sb = new StringBuilder();
48             var visited = new HashSet<ulong>();
49             links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
                ↳ renderDebug);
50             return sb.ToString();
51         }
52     }
53 }

```

```

52 public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
53     ↳ HashSet<ulong> visited, ulong linkIndex, Func<Link<ulong>, bool> isElement,
54     ↳ Action<StringBuilder, Link<ulong>> appendElement, bool renderIndex = false, bool
55     ↳ renderDebug = false)
56 {
57     if (sb == null)
58     {
59         throw new ArgumentNullException(nameof(sb));
60     }
61     if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
62     ↳ 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             }
94             sb.Append(' ');
95             if (link.Target == link.Index)
96             {
97                 sb.Append(link.Index);
98             }
99             else
100             {
101                 var target = new Link<ulong>(links.GetLink(link.Target));
102                 if (isElement(target))
103                 {
104                     appendElement(sb, target);
105                 }
106                 else
107                 {
108                     links.AppendStructure(sb, visited, target.Index, isElement,
109     ↳ appendElement, renderIndex);
110                 }
111             }
112             sb.Append(')');
113         }
114         else
115         {
116             if (renderDebug)
117             {
118                 sb.Append('*');
119             }
120             sb.Append(linkIndex);
121         }
122     }
123     else
124     {
125         if (renderDebug)
126         {
127             sb.Append('~');
128         }
129     }
130 }

```

```

124         sb.Append(linkIndex);
125     }
126 }
127 }
128 }

```

1.99 ./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         /// Или
42         ///
43         /// public struct TransitionHeader
44         /// {
45         ///     public ulong TransactionIdCombined;
46         ///     public ulong TimestampCombined;
47         ///
48         ///     public ulong TransactionId
49         ///     {
50         ///         get
51         ///         {
52         ///             return (ulong) mask & TransactionIdCombined;
53         ///         }
54         ///     }
55         ///
56         ///     public UniqueTimestamp Timestamp
57         ///     {
58         ///         get
59         ///         {
60         ///             return (UniqueTimestamp)mask & TransactionIdCombined;
61         ///         }
62         ///     }
63         ///
64         ///     public TransactionItemType Type
65         ///     {
66         ///         get
67         ///         {
68         ///             // Использовать по одному биту из TransactionId и Timestamp,
69         ///             // для значения в 2 бита, которое представляет тип операции
70         ///             throw new NotImplementedException();
71         ///         }
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
86 {
87     public static readonly long Size = Structure<Transition>.Size;
88
89     public readonly ulong TransactionId;
90     public readonly Link<ulong> Before;
91     public readonly Link<ulong> After;
92     public readonly Timestamp Timestamp;
93
94     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
95     ↪ transactionId, Link<ulong> before, Link<ulong> after)
96     {
97         TransactionId = transactionId;
98         Before = before;
99         After = after;
100        Timestamp = uniqueTimestampFactory.Create();
101    }
102
103     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
104     ↪ transactionId, Link<ulong> before)
105     : this(uniqueTimestampFactory, transactionId, before, default)
106     {
107     }
108
109     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId)
110     : this(uniqueTimestampFactory, transactionId, default, default)
111     {
112     }
113
114     public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
115     ↪ {After}";
116 }
117
118     /// <remarks>
119     /// Другие варианты реализации транзакций (атомарности):
120     /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
121     ↪ Target)) и индексов.
122     /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
123     ↪ потребуется решить вопрос
124     /// со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
125     ↪ пересечениями идентификаторов.
126     ///
127     /// Где хранить промежуточный список транзакций?
128     ///
129     /// В оперативной памяти:
130     /// Минусы:
131     /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
132     /// так как нужно отдельно выделять память под список трансформаций.
133     /// 2. Выделенной оперативной памяти может не хватить, в том случае,
134     /// если транзакция использует слишком много трансформаций.
135     /// -> Можно использовать жёсткий диск для слишком длинных транзакций.
136     /// -> Максимальный размер списка трансформаций можно ограничить / задать
137     ↪ константой.
138     /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
139     ↪ создавая задержку.
140     ///
141     /// На жёстком диске:
142     /// Минусы:
143     /// 1. Длительный отклик, на запись каждой трансформации.
144     /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
145     /// -> Это может решаться упаковкой/исключением дублирующих операций.
146     /// -> Также это может решаться тем, что короткие транзакции вообще
147     /// не будут записываться в случае отката.
148     /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
149     ↪ операции (трансформации)
150     /// будут записаны в лог.

```

```

142 ///
143 /// </remarks>
144 public class Transaction : DisposableBase
145 {
146     private readonly Queue<Transition> _transitions;
147     private readonly UInt64LinksTransactionsLayer _layer;
148     public bool IsCommitted { get; private set; }
149     public bool IsReverted { get; private set; }
150
151     public Transaction(UInt64LinksTransactionsLayer layer)
152     {
153         _layer = layer;
154         if (_layer._currentTransactionId != 0)
155         {
156             throw new NotSupportedException("Nested transactions not supported.");
157         }
158         IsCommitted = false;
159         IsReverted = false;
160         _transitions = new Queue<Transition>();
161         SetCurrentTransaction(layer, this);
162     }
163
164     public void Commit()
165     {
166         EnsureTransactionAllowsWriteOperations(this);
167         while (_transitions.Count > 0)
168         {
169             var transition = _transitions.Dequeue();
170             _layer._transitions.Enqueue(transition);
171         }
172         _layer._lastCommittedTransactionId = _layer._currentTransactionId;
173         IsCommitted = true;
174     }
175
176     private void Revert()
177     {
178         EnsureTransactionAllowsWriteOperations(this);
179         var transitionsToRevert = new Transition[_transitions.Count];
180         _transitions.CopyTo(transitionsToRevert, 0);
181         for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
182         {
183             _layer.RevertTransition(transitionsToRevert[i]);
184         }
185         IsReverted = true;
186     }
187
188     public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
189 ↪ Transaction transaction)
190     {
191         layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
192         layer._currentTransactionTransitions = transaction._transitions;
193         layer._currentTransaction = transaction;
194     }
195
196     public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
197     {
198         if (transaction.IsReverted)
199         {
200             throw new InvalidOperationException("Transation is reverted.");
201         }
202         if (transaction.IsCommitted)
203         {
204             throw new InvalidOperationException("Transation is committed.");
205         }
206     }
207
208     protected override void Dispose(bool manual, bool wasDisposed)
209     {
210         if (!wasDisposed && _layer != null && !_layer.IsDisposed)
211         {
212             if (!IsCommitted && !IsReverted)
213             {
214                 Revert();
215             }
216             _layer.ResetCurrentTransation();
217         }
218     }
219 }

```

```

220 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
221
222 private readonly string _logAddress;
223 private readonly FileStream _log;
224 private readonly Queue<Transition> _transitions;
225 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
226 private Task _transitionsPusher;
227 private Transition _lastCommittedTransition;
228 private ulong _currentTransactionId;
229 private Queue<Transition> _currentTransactionTransitions;
230 private Transaction _currentTransaction;
231 private ulong _lastCommittedTransactionId;
232
233 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
234     : base(links)
235 {
236     if (string.IsNullOrEmpty(logAddress))
237     {
238         throw new ArgumentNullException(nameof(logAddress));
239     }
240     // В первой строке файла хранится последняя законченная транзакция.
241     // При запуске это используется для проверки удачного закрытия файла лога.
242     // In the first line of the file the last committed transaction is stored.
243     // On startup, this is used to check that the log file is successfully closed.
244     var lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
245     var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
246     if (!lastCommittedTransition.Equals(lastWrittenTransition))
247     {
248         Dispose();
249         throw new NotSupportedException("Database is damaged, autorecovery is not
250             ↳ supported yet.");
251     }
252     if (lastCommittedTransition.Equals(default(Transition)))
253     {
254         FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
255     }
256     _lastCommittedTransition = lastCommittedTransition;
257     // TODO: Think about a better way to calculate or store this value
258     var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
259     _lastCommittedTransactionId = allTransitions.Length > 0 ? allTransitions.Max(x =>
260         ↳ x.TransactionId) : 0;
261     _uniqueTimestampFactory = new UniqueTimestampFactory();
262     _logAddress = logAddress;
263     _log = FileHelpers.Append(logAddress);
264     _transitions = new Queue<Transition>();
265     _transitionsPusher = new Task(TransitionsPusher);
266     _transitionsPusher.Start();
267 }
268
269 public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
270
271 public override ulong Create(IList<ulong> restrictions)
272 {
273     var createdLinkIndex = Links.Create();
274     var createdLink = new Link<ulong>(Links.GetLink(createdLinkIndex));
275     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
276         ↳ default, createdLink));
277     return createdLinkIndex;
278 }
279
280 public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
281 {
282     var linkIndex = restrictions[Constants.IndexPart];
283     var beforeLink = new Link<ulong>(Links.GetLink(linkIndex));
284     linkIndex = Links.Update(restrictions, substitution);
285     var afterLink = new Link<ulong>(Links.GetLink(linkIndex));
286     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
287         ↳ beforeLink, afterLink));
288     return linkIndex;
289 }
290
291 public override void Delete(IList<ulong> restrictions)
292 {
293     var link = restrictions[Constants.IndexPart];
294     var deletedLink = new Link<ulong>(Links.GetLink(link));
295     Links.Delete(link);
296     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
297         ↳ deletedLink, default));
298 }

```



```

294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
296     ↪ _transitions;
297
298 private void CommitTransition(Transition transition)
299 {
300     if (_currentTransaction != null)
301     {
302         Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
303     }
304     var transitions = GetCurrentTransitions();
305     transitions.Enqueue(transition);
306 }
307
308 private void RevertTransition(Transition transition)
309 {
310     if (transition.After.IsNull()) // Revert Deletion with Creation
311     {
312         Links.Create();
313     }
314     else if (transition.Before.IsNull()) // Revert Creation with Deletion
315     {
316         Links.Delete(transition.After.Index);
317     }
318     else // Revert Update
319     {
320         Links.Update(new[] { transition.After.Index, transition.Before.Source,
321             ↪ transition.Before.Target });
322     }
323 }
324
325 private void ResetCurrentTransation()
326 {
327     _currentTransactionId = 0;
328     _currentTransactionTransitions = null;
329     _currentTransaction = null;
330 }
331
332 private void PushTransitions()
333 {
334     if (_log == null || _transitions == null)
335     {
336         return;
337     }
338     for (var i = 0; i < _transitions.Count; i++)
339     {
340         var transition = _transitions.Dequeue();
341         _log.Write(transition);
342         _lastCommittedTransition = transition;
343     }
344 }
345
346 private void TransitionsPusher()
347 {
348     while (!IsDisposed && _transitionsPusher != null)
349     {
350         Thread.Sleep(DefaultPushDelay);
351         PushTransitions();
352     }
353 }
354
355 public Transaction BeginTransaction() => new Transaction(this);
356
357 private void DisposeTransitions()
358 {
359     try
360     {
361         var pusher = _transitionsPusher;
362         if (pusher != null)
363         {
364             _transitionsPusher = null;
365             pusher.Wait();
366         }
367         if (_transitions != null)
368         {
369             PushTransitions();
370         }
371     }

```

```

371         _log.DisposeIfPossible();
372         FileHelpers.WriteFirst(_logAddress, _lastCommittedTransition);
373     }
374     catch (Exception ex)
375     {
376         ex.Ignore();
377     }
378 }
379
380 #region DisposalBase
381
382 protected override void Dispose(bool manual, bool wasDisposed)
383 {
384     if (!wasDisposed)
385     {
386         DisposeTransitions();
387     }
388     base.Dispose(manual, wasDisposed);
389 }
390
391 #endregion
392 }
393 }

```

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

```

1  using Platform.Converters;
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.Unicode
7  {
8      public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
9      ↪ IConverter<char, TLink>
10     {
11         private readonly IConverter<TLink> _addressToNumberConverter;
12         private readonly TLink _unicodeSymbolMarker;
13
14         public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
15         ↪ addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
16         {
17             _addressToNumberConverter = addressToNumberConverter;
18             _unicodeSymbolMarker = unicodeSymbolMarker;
19         }
20
21         public TLink Convert(char source)
22         {
23             var unaryNumber = _addressToNumberConverter.Convert((Integer<TLink>)source);
24             return Links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
25         }
26     }
27 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Converters;
3  using Platform.Data.Doublets.Sequences.Indexes;
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 StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
10     ↪ IConverter<string, TLink>
11     {
12         private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;
13         private readonly ISequenceIndex<TLink> _index;
14         private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
15         private readonly TLink _unicodeSequenceMarker;
16
17         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
18         ↪ charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
19         ↪ TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
20         {
21             _charToUnicodeSymbolConverter = charToUnicodeSymbolConverter;
22             _index = index;
23             _listToSequenceLinkConverter = listToSequenceLinkConverter;
24             _unicodeSequenceMarker = unicodeSequenceMarker;
25         }
26     }
27 }

```

```

24     public TLink Convert(string source)
25     {
26         var elements = new TLink[source.Length];
27         for (int i = 0; i < source.Length; i++)
28         {
29             elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
30         }
31         _index.Add(elements);
32         var sequence = _listToSequenceLinkConverter.Convert(elements);
33         return Links.GetOrCreate(sequence, _unicodeSequenceMarker);
34     }
35 }
36 }

```

1.102 ./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         public UnicodeMap(ILinks<ulong> links) => _links = links;
22
23         public static UnicodeMap InitNew(ILinks<ulong> links)
24         {
25             var map = new UnicodeMap(links);
26             map.Init();
27             return map;
28         }
29
30         public void Init()
31         {
32             if (_initialized)
33             {
34                 return;
35             }
36             _initialized = true;
37             var firstLink = _links.CreatePoint();
38             if (firstLink != FirstCharLink)
39             {
40                 _links.Delete(firstLink);
41             }
42             else
43             {
44                 for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
45                 {
46                     // From NIL to It (NIL -> Character) transformation meaning, (or infinite
47                     // ↪ amount of NIL characters before actual Character)
48                     var createdLink = _links.CreatePoint();
49                     _links.Update(createdLink, firstLink, createdLink);
50                     if (createdLink != i)
51                     {
52                         throw new InvalidOperationException("Unable to initialize UTF 16
53                         ↪ table.");
54                     }
55                 }
56             }
57         }
58
59         // 0 - null link
60         // 1 - nil character (0 character)
61         // ...
62         // 65536 (0(1) + 65535 = 65536 possible values)
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         public static ulong FromCharToLink(char character) => (ulong)character + 1;

```

```

64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 public static char FromLinkToChar(ulong link) => (char)(link - 1);
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 public static bool IsCharLink(ulong link) => link <= MapSize;
69
70 public static string FromLinksToString(IList<ulong> linksList)
71 {
72     var sb = new StringBuilder();
73     for (int i = 0; i < linksList.Count; i++)
74     {
75         sb.Append(FromLinkToChar(linksList[i]));
76     }
77     return sb.ToString();
78 }
79
80 public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
81 {
82     var sb = new StringBuilder();
83     if (links.Exists(link))
84     {
85         StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
86             x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
87             ↪ element =>
88             {
89                 sb.Append(FromLinkToChar(element));
90                 return true;
91             });
92     }
93     return sb.ToString();
94 }
95
96 public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
97     ↪ chars.Length);
98
99 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
100 {
101     // char array to ulong array
102     var linksSequence = new ulong[count];
103     for (var i = 0; i < count; i++)
104     {
105         linksSequence[i] = FromCharToLink(chars[i]);
106     }
107     return linksSequence;
108 }
109
110 public static ulong[] FromStringToLinkArray(string sequence)
111 {
112     // char array to ulong array
113     var linksSequence = new ulong[sequence.Length];
114     for (var i = 0; i < sequence.Length; i++)
115     {
116         linksSequence[i] = FromCharToLink(sequence[i]);
117     }
118     return linksSequence;
119 }
120
121 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
122 {
123     var result = new List<ulong[]>();
124     var offset = 0;
125     while (offset < sequence.Length)
126     {
127         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
128         var relativeLength = 1;
129         var absoluteLength = offset + relativeLength;
130         while (absoluteLength < sequence.Length &&
131             currentCategory ==
132             ↪ CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
133         {
134             relativeLength++;
135             absoluteLength++;
136         }
137         // char array to ulong array
138         var innerSequence = new ulong[relativeLength];
139         var maxLength = offset + relativeLength;
140         for (var i = offset; i < maxLength; i++)
141         {

```

```

140         innerSequence[i - offset] = FromCharToLink(sequence[i]);
141     }
142     result.Add(innerSequence);
143     offset += relativeLength;
144 }
145 return result;
146 }
147
148 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
149 {
150     var result = new List<ulong[]>();
151     var offset = 0;
152     while (offset < array.Length)
153     {
154         var relativeLength = 1;
155         if (array[offset] <= LastCharLink)
156         {
157             var currentCategory =
158                 ↪ CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
159             var absoluteLength = offset + relativeLength;
160             while (absoluteLength < array.Length &&
161                 array[absoluteLength] <= LastCharLink &&
162                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
163                     ↪ array[absoluteLength])))
164             {
165                 relativeLength++;
166                 absoluteLength++;
167             }
168         }
169         else
170         {
171             var absoluteLength = offset + relativeLength;
172             while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
173             {
174                 relativeLength++;
175                 absoluteLength++;
176             }
177             // copy array
178             var innerSequence = new ulong[relativeLength];
179             var maxLength = offset + relativeLength;
180             for (var i = offset; i < maxLength; i++)
181             {
182                 innerSequence[i - offset] = array[i];
183             }
184             result.Add(innerSequence);
185             offset += relativeLength;
186         }
187     }
188     return result;
189 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Unicode
7 {
8     public class UnicodeSequenceCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
9         ↪ ICriterionMatcher<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13         private readonly TLink _unicodeSequenceMarker;
14         public UnicodeSequenceCriterionMatcher(ILinks<TLink> links, TLink unicodeSequenceMarker)
15             ↪ : base(links) => _unicodeSequenceMarker = unicodeSequenceMarker;
16         public bool IsMatched(TLink link) => _equalityComparer.Equals(Links.GetTarget(link),
17             ↪ _unicodeSequenceMarker);
18     }
19 }

```

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

```

1 using System;
2 using System.Linq;
3 using Platform.Interfaces;
4 using Platform.Converters;

```

```

5 using Platform.Data.Doublets.Sequences.Walkers;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Unicode
10 {
11     public class UnicodeSequenceToStringConverter<TLink> : LinksOperatorBase<TLink>,
12         ↪ IConverter<TLink, string>
13     {
14         private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
15         private readonly ISequenceWalker<TLink> _sequenceWalker;
16         private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;
17
18         public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
19             ↪ unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
20             ↪ IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
21         {
22             _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
23             _sequenceWalker = sequenceWalker;
24             _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
25         }
26
27         public string Convert(TLink source)
28         {
29             if(!_unicodeSequenceCriterionMatcher.IsMatched(source))
30             {
31                 throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
32                 ↪ not a unicode sequence.");
33             }
34             var sequence = Links.GetSource(source);
35             var charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.
36                 ↪ Convert).ToArray();
37             return new string(charArray);
38         }
39     }
40 }

```

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

```

1 using Platform.Interfaces;
2 using System.Collections.Generic;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Unicode
7 {
8     public class UnicodeSymbolCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
9         ↪ ICriterionMatcher<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13         private readonly TLink _unicodeSymbolMarker;
14         public UnicodeSymbolCriterionMatcher(ILinks<TLink> links, TLink unicodeSymbolMarker) :
15             ↪ base(links) => _unicodeSymbolMarker = unicodeSymbolMarker;
16         public bool IsMatched(TLink link) => _equalityComparer.Equals(Links.GetTarget(link),
17             ↪ _unicodeSymbolMarker);
18     }
19 }

```

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

```

1 using System;
2 using Platform.Interfaces;
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.Unicode
9 {
10     public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
11         ↪ IConverter<TLink, char>
12     {
13         private readonly IConverter<TLink> _numberToAddressConverter;
14         private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;
15
16         public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
17             ↪ numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
18             ↪ base(links)
19         {
20             _numberToAddressConverter = numberToAddressConverter;
21         }
22     }
23 }

```

```

18         _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
19     }
20
21     public char Convert(TLink source)
22     {
23         if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
24         {
25             throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                ↪ not a unicode symbol.");
26         }
27         return (char)(ushort)(Integer<TLink>)_numberToAddressConverter.Convert(Links.GetSource
                ↪ ce(source));
28     }
29 }
30 }

```

1.107 ./Platform.Data.Doublets.Tests/ComparisonTests.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Xunit;
4  using Platform.Diagnostics;
5
6  namespace Platform.Data.Doublets.Tests
7  {
8      public static class ComparisonTests
9      {
10         private class UInt64Comparer : IComparer<ulong>
11         {
12             public int Compare(ulong x, ulong y) => x.CompareTo(y);
13         }
14
15         private static int Compare(ulong x, ulong y) => x.CompareTo(y);
16
17         [Fact]
18         public static void GreaterOrEqualPerformanceTest()
19         {
20             const int N = 1000000;
21
22             ulong x = 10;
23             ulong y = 500;
24
25             bool result = false;
26
27             var ts1 = Performance.Measure(() =>
28             {
29                 for (int i = 0; i < N; i++)
30                 {
31                     result = Compare(x, y) >= 0;
32                 }
33             });
34
35             var comparer1 = Comparer<ulong>.Default;
36
37             var ts2 = Performance.Measure(() =>
38             {
39                 for (int i = 0; i < N; i++)
40                 {
41                     result = comparer1.Compare(x, y) >= 0;
42                 }
43             });
44
45             Func<ulong, ulong, int> compareReference = comparer1.Compare;
46
47             var ts3 = Performance.Measure(() =>
48             {
49                 for (int i = 0; i < N; i++)
50                 {
51                     result = compareReference(x, y) >= 0;
52                 }
53             });
54
55             var comparer2 = new UInt64Comparer();
56
57             var ts4 = Performance.Measure(() =>
58             {
59                 for (int i = 0; i < N; i++)
60                 {
61                     result = comparer2.Compare(x, y) >= 0;
62                 }
63             });

```

```

64         Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {result}");
65     }
66 }
67 }
68 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Xunit;
4  using Platform.Diagnostics;
5
6  namespace Platform.Data.Doublets.Tests
7  {
8      public static class EqualityTests
9      {
10         protected class UInt64EqualityComparer : IEqualityComparer<ulong>
11         {
12             public bool Equals(ulong x, ulong y) => x == y;
13
14             public int GetHashCode(ulong obj) => obj.GetHashCode();
15         }
16
17         private static bool Equals1<T>(T x, T y) => Equals(x, y);
18
19         private static bool Equals2<T>(T x, T y) => x.Equals(y);
20
21         private static bool Equals3(ulong x, ulong y) => x == y;
22
23         [Fact]
24         public static void EqualsPerfomanceTest()
25         {
26             const int N = 1000000;
27
28             ulong x = 10;
29             ulong y = 500;
30
31             bool result = false;
32
33             var ts1 = Performance.Measure(() =>
34             {
35                 for (int i = 0; i < N; i++)
36                 {
37                     result = Equals1(x, y);
38                 }
39             });
40
41             var ts2 = Performance.Measure(() =>
42             {
43                 for (int i = 0; i < N; i++)
44                 {
45                     result = Equals2(x, y);
46                 }
47             });
48
49             var ts3 = Performance.Measure(() =>
50             {
51                 for (int i = 0; i < N; i++)
52                 {
53                     result = Equals3(x, y);
54                 }
55             });
56
57             var equalityComparer1 = EqualityComparer<ulong>.Default;
58
59             var ts4 = Performance.Measure(() =>
60             {
61                 for (int i = 0; i < N; i++)
62                 {
63                     result = equalityComparer1.Equals(x, y);
64                 }
65             });
66
67             var equalityComparer2 = new UInt64EqualityComparer();
68
69             var ts5 = Performance.Measure(() =>
70             {
71                 for (int i = 0; i < N; i++)
72                 {
73                     result = equalityComparer2.Equals(x, y);

```



```

74     }
75 });
76
77 Func<ulong, ulong, bool> equalityComparer3 = equalityComparer2.Equals;
78
79 var ts6 = Performance.Measure(() =>
80 {
81     for (int i = 0; i < N; i++)
82     {
83         result = equalityComparer3(x, y);
84     }
85 });
86
87 var comparer = Comparer<ulong>.Default;
88
89 var ts7 = Performance.Measure(() =>
90 {
91     for (int i = 0; i < N; i++)
92     {
93         result = comparer.Compare(x, y) == 0;
94     }
95 });
96
97 Assert.True(ts2 < ts1);
98 Assert.True(ts3 < ts2);
99 Assert.True(ts5 < ts4);
100 Assert.True(ts5 < ts6);
101
102 Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {result}");
103 }
104 }
105 }

```

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

```

1  using System;
2  using Xunit;
3  using Platform.Reflection;
4  using Platform.Memory;
5  using Platform.Scopes;
6  using Platform.Data.Doublets.ResizableDirectMemory.Generic;
7
8  namespace Platform.Data.Doublets.Tests
9  {
10     public unsafe static class GenericLinksTests
11     {
12         [Fact]
13         public static void CRUDTest()
14         {
15             Using<byte>(links => links.TestCRUDOperations());
16             Using<ushort>(links => links.TestCRUDOperations());
17             Using<uint>(links => links.TestCRUDOperations());
18             Using<ulong>(links => links.TestCRUDOperations());
19         }
20
21         [Fact]
22         public static void RawNumbersCRUDTest()
23         {
24             Using<byte>(links => links.TestRawNumbersCRUDOperations());
25             Using<ushort>(links => links.TestRawNumbersCRUDOperations());
26             Using<uint>(links => links.TestRawNumbersCRUDOperations());
27             Using<ulong>(links => links.TestRawNumbersCRUDOperations());
28         }
29
30         [Fact]
31         public static void MultipleRandomCreationsAndDeletionsTest()
32         {
33             Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
34                 ↪ MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
35                 ↪ implementation of tree cuts out 5 bits from the address space.
36             Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
37                 ↪ stMultipleRandomCreationsAndDeletions(100));
38             Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
39                 ↪ MultipleRandomCreationsAndDeletions(100));
40             Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
41                 ↪ tMultipleRandomCreationsAndDeletions(100));
42         }
43
44         private static void Using<TLink>(Action<ILinks<TLink>> action)
45         {
46
47         }
48     }
49 }

```

```

41         using (var scope = new Scope<Types<HeapResizableDirectMemory,
42             ↳ ResizableDirectMemoryLinks<TLink>>>())
43         {
44             action(scope.Use<ILinks<TLink>>>());
45         }
46     }
47 }

```

1.110 ./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),
11                 ↳ (long.MaxValue + 1UL, ulong.MaxValue));
12
13             //var minimum = new Hybrid<ulong>(0, isExternal: true);
14             var minimum = new Hybrid<ulong>(1, isExternal: true);
15             var maximum = new Hybrid<ulong>(long.MaxValue, isExternal: true);
16
17             Assert.True(constants.IsExternalReference(minimum));
18             Assert.True(constants.IsExternalReference(maximum));
19         }
20     }

```

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

```

1  using System;
2  using System.Linq;
3  using Xunit;
4  using Platform.Memory;
5  using Platform.Data.Numbers.Raw;
6  using Platform.Data.Doublets.Sequences;
7  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
8  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
9  using Platform.Data.Doublets.Sequences.Converters;
10 using Platform.Data.Doublets.PropertyOperators;
11 using Platform.Data.Doublets.Incrementers;
12 using Platform.Data.Doublets.Sequences.Walkers;
13 using Platform.Data.Doublets.Sequences.Indexes;
14 using Platform.Data.Doublets.Unicode;
15 using Platform.Data.Doublets.Numbers.Unary;
16 using Platform.Data.Doublets.Decorators;
17 using Platform.Data.Doublets.ResizableDirectMemory.Specific;
18 using Platform.Collections.Stacks;
19
20 namespace Platform.Data.Doublets.Tests
21 {
22     public static class OptimalVariantSequenceTests
23     {
24         private static readonly string _sequenceExample = "зеленела зелёная зелень";
25         private static readonly string _loremIpsumExample = @"Lorem ipsum dolor sit amet,
26             ↳ consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore
27             ↳ magna aliqua.
28             Facilisi nullam vehicula ipsum a arcu cursus vitae congue mauris.
29             Et malesuada fames ac turpis egestas sed.
30             Eget velit aliquet sagittis id consectetur purus.
31             Dignissim cras tincidunt lobortis feugiat vivamus.
32             Vitae aliquet nec ullamcorper sit.
33             Lectus quam id leo in vitae.
34             Tortor dignissim convallis aenean et tortor at risus viverra adipiscing.
35             Sed risus ultricies tristique nulla aliquet enim tortor at auctor.
36             Integer eget aliquet nibh praesent tristique.
37             Vitae congue eu consequat ac felis donec et odio.
38             Tristique et egestas quis ipsum suspendisse.
39             Suspendisse potenti nullam ac tortor vitae purus faucibus ornare.
40             Nulla facilisi etiam dignissim diam quis enim lobortis scelerisque.
41             Imperdiet proin fermentum leo vel orci.
42             In ante metus dictum at tempor commodo.
43             Nisi lacus sed viverra tellus in.
44             Quam vulputate dignissim suspendisse in.
45             Elit scelerisque mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus.
46             Gravida cum sociis natoque penatibus et magnis dis parturient.
47             Risus quis varius quam quisque id diam.
48             Congue nisliv vitae suscipit tellus mauris a diam maecenas.
49             Eget nunc scelerisque viverra mauris in aliquam sem fringilla.

```

```

48 Pharetra vel turpis nunc eget lorem dolor sed viverra.
49 Mattis pellentesque id nibh tortor id aliquet.
50 Purus non enim praesent elementum facilisis leo vel.
51 Etiam sit amet nisl purus in mollis nunc sed.
52 Tortor at auctor urna nunc id cursus metus aliquam.
53 Volutpat odio facilisis mauris sit amet.
54 Turpis egestas pretium aenean pharetra magna ac placerat.
55 Fermentum dui faucibus in ornare quam viverra orci sagittis eu.
56 Porttitor leo a diam sollicitudin tempor id eu.
57 Volutpat sed cras ornare arcu dui.
58 Ut aliquam purus sit amet luctus venenatis lectus magna.
59 Aliquet risus feugiat in ante metus dictum at.
60 Mattis nunc sed blandit libero.
61 Elit pellentesque habitant morbi tristique senectus et netus.
62 Nibh sit amet commodo nulla facilisi nullam vehicula ipsum a.
63 Enim sit amet venenatis urna cursus eget nunc scelerisque viverra.
64 Amet venenatis urna cursus eget nunc scelerisque viverra mauris in.
65 Diam donec adipiscing tristique risus nec feugiat.
66 Pulvinar mattis nunc sed blandit libero volutpat.
67 Cras fermentum odio eu feugiat pretium nibh ipsum.
68 In nulla posuere sollicitudin aliquam ultrices sagittis orci a.
69 Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.
70 A iaculis at erat pellentesque.
71 Morbi blandit cursus risus at ultrices mi tempus imperdiet nulla.
72 Eget lorem dolor sed viverra ipsum nunc.
73 Leo a diam sollicitudin tempor id eu.
74 Interdum consectetur libero id faucibus nisl tincidunt eget nullam non.";
75
76 [Fact]
77 public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
78 {
79     using (var scope = new TempLinksTestScope(useSequences: false))
80     {
81         var links = scope.Links;
82         var constants = links.Constants;
83
84         links.UseUnicode();
85
86         var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
87
88         var meaningRoot = links.CreatePoint();
89         var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
90         var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
91         var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
92             ↪ constants.Itself);
93
94         var unaryNumberToAddressConverter = new
95             ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
96         var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
97         var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
98             ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
99         var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
100             ↪ frequencyPropertyMarker, frequencyMarker);
101         var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
102             ↪ frequencyPropertyOperator, frequencyIncrementer);
103         var linkToItsFrequencyNumberConverter = new
104             ↪ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
105             ↪ unaryNumberToAddressConverter);
106         var sequenceToItsLocalElementLevelsConverter = new
107             ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
108             ↪ linkToItsFrequencyNumberConverter);
109         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
110             ↪ sequenceToItsLocalElementLevelsConverter);
111
112         var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
113             ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
114
115         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
116             ↪ index, optimalVariantConverter);
117     }
118 }
119
120 [Fact]
121 public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
122 {
123     using (var scope = new TempLinksTestScope(useSequences: false))
124     {
125         var links = scope.Links;
126
127         links.UseUnicode();

```

```

116     var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
117
118     var totalSequenceSymbolFrequencyCounter = new
119         ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links);
120
121     var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
122         ↳ totalSequenceSymbolFrequencyCounter);
123
124     var index = new
125         ↳ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
126     var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
127
128     var sequenceToItsLocalElementLevelsConverter = new
129         ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
130         ↳ linkToItsFrequencyNumberConverter);
131     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
132         ↳ sequenceToItsLocalElementLevelsConverter);
133
134     var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
135         ↳ Walker = new LeveledSequenceWalker<ulong>(links) });
136
137     ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
138         ↳ index, optimalVariantConverter);
139 }
140
141 private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
142     ↳ SequenceToItsLocalElementLevelsConverter<ulong>
143     ↳ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
144     ↳ OptimalVariantConverter<ulong> optimalVariantConverter)
145 {
146     index.Add(sequence);
147
148     var optimalVariant = optimalVariantConverter.Convert(sequence);
149
150     var readSequence1 = sequences.ToList(optimalVariant);
151
152     Assert.True(sequence.SequenceEqual(readSequence1));
153 }
154
155 [Fact]
156 public static void SavedSequencesOptimizationTest()
157 {
158     LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
159         ↳ (long.MaxValue + 1UL, ulong.MaxValue));
160
161     using (var memory = new HeapResizableDirectMemory())
162     using (var disposableLinks = new UInt64ResizableDirectMemoryLinks(memory,
163         ↳ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep, constants,
164         ↳ useAvlBasedIndex: false))
165     {
166         var links = new UInt64Links(disposableLinks);
167
168         var root = links.CreatePoint();
169
170         //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
171         var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
172
173         var unicodeSymbolMarker = links.GetOrCreate(root,
174             ↳ addressToNumberConverter.Convert(1));
175         var unicodeSequenceMarker = links.GetOrCreate(root,
176             ↳ addressToNumberConverter.Convert(2));
177
178         var totalSequenceSymbolFrequencyCounter = new
179             ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links);
180         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
181             ↳ totalSequenceSymbolFrequencyCounter);
182         var index = new
183             ↳ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
184         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
185         var sequenceToItsLocalElementLevelsConverter = new
186             ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
187             ↳ linkToItsFrequencyNumberConverter);
188         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
189             ↳ sequenceToItsLocalElementLevelsConverter);

```

```

170
171     var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
172         ↪ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
173
174     var unicodeSequencesOptions = new SequencesOptions<ulong>()
175     {
176         UseSequenceMarker = true,
177         SequenceMarkerLink = unicodeSequenceMarker,
178         UseIndex = true,
179         Index = index,
180         LinksToSequenceConverter = optimalVariantConverter,
181         Walker = walker,
182         UseGarbageCollection = true
183     };
184
185     var unicodeSequences = new Sequences.Sequences(new
186         ↪ SynchronizedLinks<ulong>(links), unicodeSequencesOptions);
187
188     // Create some sequences
189     var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
190         ↪ StringSplitOptions.RemoveEmptyEntries);
191     var arrays = strings.Select(x => x.Select(y =>
192         ↪ addressToNumberConverter.Convert(y)).ToArray()).ToArray();
193     for (int i = 0; i < arrays.Length; i++)
194     {
195         unicodeSequences.Create(arrays[i].ConvertToRestrictionsValues());
196     }
197
198     var linksCountAfterCreation = links.Count();
199
200     // get list of sequences links
201     // for each sequence link
202     //     create new sequence version
203     //     if new sequence is not the same as sequence link
204     //         delete sequence link
205     //         collect garbadge
206     unicodeSequences.CompactAll();
207
208     var linksCountAfterCompactification = links.Count();
209
210     Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
211 }
212 }
213 }
214 }
215 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Linq;
5  using Xunit;
6  using Platform.Data.Sequences;
7  using Platform.Data.Doublets.Sequences.Converters;
8  using Platform.Data.Doublets.Sequences.Walkers;
9  using Platform.Data.Doublets.Sequences;
10
11 namespace Platform.Data.Doublets.Tests
12 {
13     public static class ReadSequenceTests
14     {
15         [Fact]
16         public static void ReadSequenceTest()
17         {
18             const long sequenceLength = 2000;
19
20             using (var scope = new TempLinksTestScope(useSequences: false))
21             {
22                 var links = scope.Links;
23                 var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong> {
24                     ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
25
26                 var sequence = new ulong[sequenceLength];
27                 for (var i = 0; i < sequenceLength; i++)
28                 {
29                     sequence[i] = links.Create();
30                 }
31
32                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);

```

```

33     var sw1 = Stopwatch.StartNew();
34     var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
35
36     var sw2 = Stopwatch.StartNew();
37     var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();
38
39     var sw3 = Stopwatch.StartNew();
40     var readSequence2 = new List<ulong>();
41     SequenceWalker.WalkRight(balancedVariant,
42                             links.GetSource,
43                             links.GetTarget,
44                             links.IsPartialPoint,
45                             readSequence2.Add);
46
47     sw3.Stop();
48
49     Assert.True(sequence.SequenceEqual(readSequence1));
50
51     Assert.True(sequence.SequenceEqual(readSequence2));
52
53     // Assert.True(sw2.Elapsed < sw3.Elapsed);
54
55     Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
56     ↪ {sw2.Elapsed}");
57
58     for (var i = 0; i < sequenceLength; i++)
59     {
60         links.Delete(sequence[i]);
61     }
62 }
63 }

```

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

```

1  using System.IO;
2  using Xunit;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using Platform.Data.Doublets.ResizableDirectMemory.Specific;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public static class ResizableDirectMemoryLinksTests
10     {
11         private static readonly LinksConstants<ulong> _constants =
12             ↪ Default<LinksConstants<ulong>>.Instance;
13
14         [Fact]
15         public static void BasicFileMappedMemoryTest()
16         {
17             var tempFilename = Path.GetTempFileName();
18             using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(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(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
29                 using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
30                     ↪ UInt64ResizableDirectMemoryLinks.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
41                 [Fact]
42                 public static void NonexistentReferencesHeapMemoryTest()
43                 {

```

```

43     using (var memory = new
44         ↪ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
45     using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
46         ↪ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
47     {
48         memoryAdapter.TestNonexistentReferences();
49     }
50 private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
51 {
52     var link = memoryAdapter.Create();
53     memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
54     var resultLink = _constants.Null;
55     memoryAdapter.Each(foundLink =>
56     {
57         resultLink = foundLink[_constants.IndexPart];
58         return _constants.Break;
59     }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
60     Assert.True(resultLink == link);
61     Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
62     memoryAdapter.Delete(link);
63 }
64 }
65 }

```

1.114 ./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.ResizableDirectMemory.Generic;
7  using Platform.Data.Doublets.ResizableDirectMemory.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<UInt64ResizableDirectMemoryLinks>();
31                 var instance = scope.Use<ILinks<ulong>>();
32                 Assert.IsType<UInt64ResizableDirectMemoryLinks>(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                 ↪ ResizableDirectMemoryLinks<ulong>>>())
51             {
52                 var links = scope.Use<ILinks<ulong>>();

```

```

52         Assert.IsType<ResizableDirectMemoryLinks<ulong>>(links);
53     }
54 }
55 }
56 }

```

1.115 ./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.Random;
8  using Platform.IO;
9  using Platform.Singletons;
10 using Platform.Data.Doublets.Sequences;
11 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
12 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
13 using Platform.Data.Doublets.Sequences.Converters;
14 using Platform.Data.Doublets.Unicode;
15
16 namespace Platform.Data.Doublets.Tests
17 {
18     public static class SequencesTests
19     {
20         private static readonly LinksConstants<ulong> _constants =
21             ↪ Default<LinksConstants<ulong>>.Instance;
22
23         static SequencesTests()
24         {
25             // Trigger static constructor to not mess with performance measurements
26             _ = BitString.GetBitMaskFromIndex(1);
27         }
28
29         [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             //[Fact]
63             //public void CUDDTest()
64             //{
65             //    var tempFilename = Path.GetTempFileName();
66
67             //    const long sequenceLength = 8;
68
69             //    const ulong itself = LinksConstants.Itself;
70
71             //    using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
72             //        ↪ DefaultLinksSizeStep))
73             //    using (var links = new Links(memoryAdapter))

```



```

73 // {
74 //     var sequence = new ulong[sequenceLength];
75 //     for (var i = 0; i < sequenceLength; i++)
76 //         sequence[i] = links.Create(itself, itself);
77
78 //     SequencesOptions o = new SequencesOptions();
79
80 // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
81 //     o.
82
83 //     var sequences = new Sequences(links);
84
85 //     var sw1 = Stopwatch.StartNew();
86 //     var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
87
88 //     var sw2 = Stopwatch.StartNew();
89 //     var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
90
91 //     Assert.True(results1.Count > results2.Length);
92 //     Assert.True(sw1.Elapsed > sw2.Elapsed);
93
94 //     for (var i = 0; i < sequenceLength; i++)
95 //         links.Delete(sequence[i]);
96 // }
97
98 // File.Delete(tempFilename);
99 //}

```

```

100
101 [Fact]
102 public static void AllVariantsSearchTest()
103 {
104     const long sequenceLength = 8;
105
106     using (var scope = new TempLinksTestScope(useSequences: true))
107     {
108         var links = scope.Links;
109         var sequences = scope.Sequences;
110
111         var sequence = new ulong[sequenceLength];
112         for (var i = 0; i < sequenceLength; i++)
113         {
114             sequence[i] = links.Create();
115         }
116
117         var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
118
119         //for (int i = 0; i < createResults.Length; i++)
120         //    sequences.Create(createResults[i]);
121
122         var sw0 = Stopwatch.StartNew();
123         var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
124
125         var sw1 = Stopwatch.StartNew();
126         var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
127
128         var sw2 = Stopwatch.StartNew();
129         var searchResults2 = sequences.Each1(sequence); sw2.Stop();
130
131         var sw3 = Stopwatch.StartNew();
132         var searchResults3 = sequences.Each(sequence.ConvertToRestrictionsValues());
133         ↪ 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         {

```

```

152         links.Delete(sequence[i]);
153     }
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         Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
189
190         Assert.True(searchResults3.Count == 1 && balancedVariant ==
191             ↪ searchResults3.First());
192
193         //Assert.True(sw1.Elapsed < sw2.Elapsed);
194
195         for (var i = 0; i < sequenceLength; i++)
196         {
197             links.Delete(sequence[i]);
198         }
199     }
200 }
201
202 [Fact]
203 public static void AllPartialVariantsSearchTest()
204 {
205     const long sequenceLength = 8;
206
207     using (var scope = new TempLinksTestScope(useSequences: true))
208     {
209         var links = scope.Links;
210         var sequences = scope.Sequences;
211
212         var sequence = new ulong[sequenceLength];
213         for (var i = 0; i < sequenceLength; i++)
214         {
215             sequence[i] = links.Create();
216         }
217
218         var createResults = sequences.CreateAllVariants2(sequence);
219
220         //var createResultsStrings = createResults.Select(x => x + ": " +
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();

```

```

229     var sw2 = Stopwatch.StartNew();
230     var searchResults2 =
231         ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
232
233     //var sw3 = Stopwatch.StartNew();
234     //var searchResults3 =
235         ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
236
237     var sw4 = Stopwatch.StartNew();
238     var searchResults4 =
239         ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
240
241     //Global.Trash = searchResults3;
242
243     //var searchResults1Strings = searchResults1.Select(x => x + ": " +
244         ↪ sequences.FormatSequence(x)).ToList();
245     //Global.Trash = searchResults1Strings;
246
247     var intersection1 = createResults.Intersect(searchResults1).ToList();
248     Assert.True(intersection1.Count == createResults.Length);
249
250     var intersection2 = createResults.Intersect(searchResults2).ToList();
251     Assert.True(intersection2.Count == createResults.Length);
252
253     var intersection4 = createResults.Intersect(searchResults4).ToList();
254     Assert.True(intersection4.Count == createResults.Length);
255
256     for (var i = 0; i < sequenceLength; i++)
257     {
258         links.Delete(sequence[i]);
259     }
260 }
261
262 [Fact]
263 public static void BalancedPartialVariantsSearchTest()
264 {
265     const long sequenceLength = 200;
266
267     using (var scope = new TempLinksTestScope(useSequences: true))
268     {
269         var links = scope.Links;
270         var sequences = scope.Sequences;
271
272         var sequence = new ulong[sequenceLength];
273         for (var i = 0; i < sequenceLength; i++)
274         {
275             sequence[i] = links.Create();
276         }
277
278         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
279         var balancedVariant = balancedVariantConverter.Convert(sequence);
280
281         var partialSequence = new ulong[sequenceLength - 2];
282         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
283
284         var sw1 = Stopwatch.StartNew();
285         var searchResults1 =
286             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
287
288         var sw2 = Stopwatch.StartNew();
289         var searchResults2 =
290             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
291
292         Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
293
294         Assert.True(searchResults2.Count == 1 && balancedVariant ==
295             ↪ searchResults2.First());
296
297         for (var i = 0; i < sequenceLength; i++)
298         {
299             links.Delete(sequence[i]);
300         }
301     }
302 }
303
304 [Fact(Skip = "Correct implementation is pending")]

```

```

301 public static void PatternMatchTest()
302 {
303     var zeroOrMany = Sequences.Sequences.ZeroOrMany;
304
305     using (var scope = new TempLinksTestScope(useSequences: true))
306     {
307         var links = scope.Links;
308         var sequences = scope.Sequences;
309
310         var e1 = links.Create();
311         var e2 = links.Create();
312
313         var sequence = new[]
314         {
315             e1, e2, e1, e2 // mama / papa
316         };
317
318         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
319
320         var balancedVariant = balancedVariantConverter.Convert(sequence);
321
322         // 1: [1]
323         // 2: [2]
324         // 3: [1,2]
325         // 4: [1,2,1,2]
326
327         var doublet = links.GetSource(balancedVariant);
328
329         var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
330
331         Assert.True(matchedSequences1.Count == 0);
332
333         var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
334
335         Assert.True(matchedSequences2.Count == 0);
336
337         var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
338
339         Assert.True(matchedSequences3.Count == 0);
340
341         var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
342
343         Assert.Contains(doublet, matchedSequences4);
344         Assert.Contains(balancedVariant, matchedSequences4);
345
346         for (var i = 0; i < sequence.Length; i++)
347         {
348             links.Delete(sequence[i]);
349         }
350     }
351 }
352
353 [Fact]
354 public static void IndexTest()
355 {
356     using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
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/%D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87%D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
```

```
379 private static readonly string _exampleText =  
380     @"([english  
381         ↪ version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
```

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

```
383 [[чёрное пространство, белое  
384     пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png  
385     ↪ ""чёрное пространство, белое пространство"")](https://raw.githubusercontent.com/Konard/Links  
386     Platform/master/doc/Intro/1.png)
```

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

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

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

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

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

```
395 [[две белые точки, чёрная вертикальная  
396     линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png ""две  
397     ↪ белые точки, чёрная вертикальная  
398     линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)
```

397 Можно ли замкнуть движение? Может ли это быть кругом? Можно ли замкнуть время? Или остаётся
398 ↪ только спираль? Но что если замкнуть предел? Создать ограничение, разделение? Получится
399 ↪ замкнутая область? Полностью отделённая от всего остального? Но что это всё остальное? Что
400 ↪ можно делить? В каком направлении? Ничего или всё? Пустота или полнота? Начало или конец?
401 ↪ Или может быть это единица и ноль? Дуальность? Противоположность? А что будет с кругом если
402 ↪ у него нет размера? Будет ли круг точкой? Точка состоящая из точек?

```
399 [[белая вертикальная линия, чёрный  
400     круг](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png ""белая  
401     ↪ вертикальная линия, чёрный  
402     круг"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png)
```

401 Как ещё можно использовать грань, черту, линию? А что если она может что-то соединять, может
402 ↪ тогда её нужно повернуть? Почему то, что перпендикулярно вертикальному горизонтально?
403 ↪ Горизонт? Инвертирует ли это смысл? Что такое смысл? Из чего состоит смысл? Существует ли
404 ↪ элементарная единица смысла?

```
403 [[белый круг, чёрная горизонтальная  
404     линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png ""белый  
405     ↪ круг, чёрная горизонтальная  
406     линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png)
```

405 Соединять, допустим, а какой смысл в этом есть ещё? Что если помимо смысла ""соединить,
406 ↪ связать"", есть ещё и смысл направления ""от начала к концу""? От предка к потомку? От
407 ↪ родителя к ребёнку? От общего к частному?

```
407 [[белая горизонтальная линия, чёрная горизонтальная  
408     стрелка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png  
409     ↪ ""белая горизонтальная линия, чёрная горизонтальная  
410     ↪ стрелка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png)
```

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

```

412 [![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)
413
414 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?
415
416 [![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)
417
418 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?
419
420 [![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)
421
422 На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
↳ рекурсии или фрактала?
423
424 [![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)
425
426 Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
↳ Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
427
428 [![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)
429
430 ...
431
432 [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animat
↳ ion-500.gif
↳ "анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
↳ -animation-500.gif)";
433
434     private static readonly string _exampleLoremIpsumText =
435         @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
↳ incididunt ut labore et dolore magna aliqua.
436 Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
↳ consequat.";
437
438 [Fact]
439 public static void CompressionTest()
440 {
441     using (var scope = new TempLinksTestScope(useSequences: true))
442     {
443         var links = scope.Links;
444         var sequences = scope.Sequences;
445
446         var e1 = links.Create();
447         var e2 = links.Create();
448
449         var sequence = new[]
450         {
451             e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
452         };
453
454         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
455         var totalSequenceSymbolFrequencyCounter = new
↳ TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);

```

```

456     var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
457         ↪ totalSequenceSymbolFrequencyCounter);
458
459     var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
460         ↪ balancedVariantConverter, doubletFrequenciesCache);
461
462     var compressedVariant = compressingConverter.Convert(sequence);
463
464     // 1: [1]          (1->1) point
465     // 2: [2]          (2->2) point
466     // 3: [1,2]        (1->2) doublet
467     // 4: [1,2,1,2]    (3->3) doublet
468
469     Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
470     Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
471     Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
472     Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
473
474     var source = _constants.SourcePart;
475     var target = _constants.TargetPart;
476
477     Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
478     Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
479     Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
480     Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
481
482     // 4 - length of sequence
483     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
484         ↪ == sequence[0]);
485     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
486         ↪ == sequence[1]);
487     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
488         ↪ == sequence[2]);
489     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
490         ↪ == sequence[3]);
491 }
492
493 [Fact]
494 public static void CompressionEfficiencyTest()
495 {
496     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
497         ↪ StringSplitOptions.RemoveEmptyEntries);
498     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
499     var totalCharacters = arrays.Select(x => x.Length).Sum();
500
501     using (var scope1 = new TempLinksTestScope(useSequences: true))
502     using (var scope2 = new TempLinksTestScope(useSequences: true))
503     using (var scope3 = new TempLinksTestScope(useSequences: true))
504     {
505         scope1.Links.Unsync.UseUnicode();
506         scope2.Links.Unsync.UseUnicode();
507         scope3.Links.Unsync.UseUnicode();
508
509         var balancedVariantConverter1 = new
510             ↪ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
511         var totalSequenceSymbolFrequencyCounter = new
512             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
513         var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
514             ↪ totalSequenceSymbolFrequencyCounter);
515         var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
516             ↪ balancedVariantConverter1, linkFrequenciesCache1,
517             ↪ doInitialFrequenciesIncrement: false);
518
519         //var compressor2 = scope2.Sequences;
520         var compressor3 = scope3.Sequences;
521
522         var constants = Default<LinksConstants<ulong>>.Instance;
523
524         var sequences = compressor3;
525         //var meaningRoot = links.CreatePoint();
526         //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
527         //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
528         //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
529             ↪ constants.Itself);
530
531         //var unaryNumberToAddressConverter = new
532             ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);

```

```

519 //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
    ↳ unaryOne);
520 //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
    ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
521 //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
    ↳ frequencyPropertyMarker, frequencyMarker);
522 //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
    ↳ frequencyPropertyOperator, frequencyIncrementer);
523 //var linkToItsFrequencyNumberConverter = new
    ↳ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
    ↳ unaryNumberToAddressConverter);
524
525 var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
    ↳ totalSequenceSymbolFrequencyCounter);
526
527 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache3);
528
529 var sequenceToItsLocalElementLevelsConverter = new
    ↳ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
    ↳ linkToItsFrequencyNumberConverter);
530 var optimalVariantConverter = new
    ↳ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
    ↳ sequenceToItsLocalElementLevelsConverter);
531
532 var compressed1 = new ulong[arrays.Length];
533 var compressed2 = new ulong[arrays.Length];
534 var compressed3 = new ulong[arrays.Length];
535
536 var START = 0;
537 var END = arrays.Length;
538
539 //for (int i = START; i < END; i++)
540 //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
541
542 var initialCount1 = scope2.Links.Unsync.Count();
543
544 var sw1 = Stopwatch.StartNew();
545
546 for (int i = START; i < END; i++)
547 {
548     linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
549     compressed1[i] = compressor1.Convert(arrays[i]);
550 }
551
552 var elapsed1 = sw1.Elapsed;
553
554 var balancedVariantConverter2 = new
    ↳ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
555
556 var initialCount2 = scope2.Links.Unsync.Count();
557
558 var sw2 = Stopwatch.StartNew();
559
560 for (int i = START; i < END; i++)
561 {
562     compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
563 }
564
565 var elapsed2 = sw2.Elapsed;
566
567 for (int i = START; i < END; i++)
568 {
569     linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
570 }
571
572 var initialCount3 = scope3.Links.Unsync.Count();
573
574 var sw3 = Stopwatch.StartNew();
575
576 for (int i = START; i < END; i++)
577 {
578     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
579     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
580 }
581
582 var elapsed3 = sw3.Elapsed;
583

```



```

584 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
    ↳ Optimal variant: {elapsed3}");
585
586 // Assert.True(elapsed1 > elapsed2);
587
588 // Checks
589 for (int i = START; i < END; i++)
590 {
591     var sequence1 = compressed1[i];
592     var sequence2 = compressed2[i];
593     var sequence3 = compressed3[i];
594
595     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
    ↳ scope1.Links.Unsync);
596
597     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
    ↳ scope2.Links.Unsync);
598
599     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
    ↳ scope3.Links.Unsync);
600
601     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
    ↳ link.IsPartialPoint());
602     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
    ↳ link.IsPartialPoint());
603     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
    ↳ link.IsPartialPoint());
604
605     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
    ↳ arrays[i].Length > 3)
606     //    Assert.False(structure1 == structure2);
607     //if (sequence2 != Constants.Null && sequence3 != Constants.Null &&
    ↳ arrays[i].Length > 3)
608     //    Assert.False(structure3 == structure2);
609
610     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
611     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
612 }
613
614 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
    ↳ totalCharacters);
615 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
    ↳ totalCharacters);
616 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
    ↳ totalCharacters);
617
618 Console.WriteLine($"{{(double)(scope1.Links.Unsync.Count() - initialCount1) /
    ↳ totalCharacters}} | {{(double)(scope2.Links.Unsync.Count() - initialCount2) /
    ↳ totalCharacters}} | {{(double)(scope3.Links.Unsync.Count() - initialCount3) /
    ↳ totalCharacters}}");
619
620 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
    ↳ scope2.Links.Unsync.Count() - initialCount2);
621 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
    ↳ scope2.Links.Unsync.Count() - initialCount2);
622
623 var duplicateProvider1 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
624 var duplicateProvider2 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
625 var duplicateProvider3 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
626
627 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
628 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
629 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
630
631 var duplicates1 = duplicateCounter1.Count();
632
633 ConsoleHelpers.Debug("-----");
634
635 var duplicates2 = duplicateCounter2.Count();
636
637 ConsoleHelpers.Debug("-----");
638
639 var duplicates3 = duplicateCounter3.Count();
640
641 Console.WriteLine($"{{duplicates1}} | {{duplicates2}} | {{duplicates3}}");

```

```

642         linkFrequenciesCache1.ValidateFrequencies();
643         linkFrequenciesCache3.ValidateFrequencies();
644     }
645 }
646
647 [Fact]
648 public static void CompressionStabilityTest()
649 {
650     // TODO: Fix bug (do a separate test)
651     //const ulong minNumbers = 0;
652     //const ulong maxNumbers = 1000;
653
654     const ulong minNumbers = 10000;
655     const ulong maxNumbers = 12500;
656
657     var strings = new List<string>();
658
659     for (ulong i = minNumbers; i < maxNumbers; i++)
660     {
661         strings.Add(i.ToString());
662     }
663
664     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
665     var totalCharacters = arrays.Select(x => x.Length).Sum();
666
667     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
668         ↪ SequencesOptions<ulong> { UseCompression = true,
669         ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
670     using (var scope2 = new TempLinksTestScope(useSequences: true))
671     {
672         scope1.Links.UseUnicode();
673         scope2.Links.UseUnicode();
674
675         //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
676         var compressor1 = scope1.Sequences;
677         var compressor2 = scope2.Sequences;
678
679         var compressed1 = new ulong[arrays.Length];
680         var compressed2 = new ulong[arrays.Length];
681
682         var sw1 = Stopwatch.StartNew();
683
684         var START = 0;
685         var END = arrays.Length;
686
687         // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
688         // Stability issue starts at 10001 or 11000
689         //for (int i = START; i < END; i++)
690         //{
691         //    var first = compressor1.Compress(arrays[i]);
692         //    var second = compressor1.Compress(arrays[i]);
693
694         //    if (first == second)
695         //        compressed1[i] = first;
696         //    else
697         //    {
698         //        // TODO: Find a solution for this case
699         //    }
700         //}
701
702         for (int i = START; i < END; i++)
703         {
704             var first = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
705             var second = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
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     var sw2 = Stopwatch.StartNew();
721
722     for (int i = START; i < END; i++)
723     {
724         var first = balancedVariantConverter.Convert(arrays[i]);
725         var second = balancedVariantConverter.Convert(arrays[i]);
726
727         if (first == second)
728         {
729             compressed2[i] = first;
730         }
731     }
732
733     var elapsed2 = sw2.Elapsed;
734
735     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
736     ↪ {elapsed2}");
737
738     Assert.True(elapsed1 > elapsed2);
739
740     // Checks
741     for (int i = START; i < END; i++)
742     {
743         var sequence1 = compressed1[i];
744         var sequence2 = compressed2[i];
745
746         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
747         {
748             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
749             ↪ scope1.Links);
750
751             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
752             ↪ scope2.Links);
753
754             //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
755             ↪ link.IsPartialPoint());
756             //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
757             ↪ link.IsPartialPoint());
758
759             //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
760             ↪ arrays[i].Length > 3)
761             //    Assert.False(structure1 == structure2);
762
763             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
764         }
765     }
766
767     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
768     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
769
770     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
771     ↪ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
772     ↪ totalCharacters}}");
773
774     Assert.True(scope1.Links.Count() <= scope2.Links.Count());
775
776     //compressor1.ValidateFrequencies();
777 }
778
779 [Fact]
780 public static void RandomNumbersCompressionQualityTest()
781 {
782     const ulong N = 500;
783
784     //const ulong minNumbers = 10000;
785     //const ulong maxNumbers = 20000;
786
787     //var strings = new List<string>();
788
789     //for (ulong i = 0; i < N; i++)
790     //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
791     ↪ maxNumbers).ToString());
792
793     var strings = new List<string>();
794
795     for (ulong i = 0; i < N; i++)
796     {
797         strings.Add(RandomHelpers.Default.NextUInt64().ToString());
798     }
799 }

```

```

790 }
791
792 strings = strings.Distinct().ToList();
793
794 var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
795 var totalCharacters = arrays.Select(x => x.Length).Sum();
796
797 using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
    ↳ SequencesOptions<ulong> { UseCompression = true,
    ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
using (var scope2 = new TempLinksTestScope(useSequences: true))
{
    scope1.Links.UseUnicode();
    scope2.Links.UseUnicode();

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

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

    var sw1 = Stopwatch.StartNew();

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

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

    var elapsed1 = sw1.Elapsed;

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

    var sw2 = Stopwatch.StartNew();

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

    var elapsed2 = sw2.Elapsed;

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

    Assert.True(elapsed1 > elapsed2);

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

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

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

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

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

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

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

    //compressor1.ValidateFrequencies();
}

```

```

862 }
863
864 [Fact]
865 public static void AllTreeBreakDownAtSequencesCreationBugTest()
866 {
867     // Made out of AllPossibleConnectionsTest test.
868
869     //const long sequenceLength = 5; //100% bug
870     const long sequenceLength = 4; //100% bug
871     //const long sequenceLength = 3; //100% _no_bug (ok)
872
873     using (var scope = new TempLinksTestScope(useSequences: true))
874     {
875         var links = scope.Links;
876         var sequences = scope.Sequences;
877
878         var sequence = new ulong[sequenceLength];
879         for (var i = 0; i < sequenceLength; i++)
880         {
881             sequence[i] = links.Create();
882         }
883
884         var createResults = sequences.CreateAllVariants2(sequence);
885         Global.Trash = createResults;
886
887         for (var i = 0; i < sequenceLength; i++)
888         {
889             links.Delete(sequence[i]);
890         }
891     }
892 }
893
894 [Fact]
895 public static void AllPossibleConnectionsTest()
896 {
897     const long sequenceLength = 5;
898
899     using (var scope = new TempLinksTestScope(useSequences: true))
900     {
901         var links = scope.Links;
902         var sequences = scope.Sequences;
903
904         var sequence = new ulong[sequenceLength];
905         for (var i = 0; i < sequenceLength; i++)
906         {
907             sequence[i] = links.Create();
908         }
909
910         var createResults = sequences.CreateAllVariants2(sequence);
911         var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
912
913         for (var i = 0; i < 1; i++)
914         {
915             var sw1 = Stopwatch.StartNew();
916             var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
917
918             var sw2 = Stopwatch.StartNew();
919             var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
920
921             var sw3 = Stopwatch.StartNew();
922             var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
923
924             var sw4 = Stopwatch.StartNew();
925             var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
926
927             Global.Trash = searchResults3;
928             Global.Trash = searchResults4; //-V3008
929
930             var intersection1 = createResults.Intersect(searchResults1).ToList();
931             Assert.True(intersection1.Count == createResults.Length);
932
933             var intersection2 = reverseResults.Intersect(searchResults1).ToList();
934             Assert.True(intersection2.Count == reverseResults.Length);
935
936             var intersection0 = searchResults1.Intersect(searchResults2).ToList();
937             Assert.True(intersection0.Count == searchResults2.Count);
938
939             var intersection3 = searchResults2.Intersect(searchResults3).ToList();
940             Assert.True(intersection3.Count == searchResults3.Count);
941

```

```

942         var intersection4 = searchResults3.Intersect(searchResults4).ToList();
943         Assert.True(intersection4.Count == searchResults4.Count);
944     }
945
946     for (var i = 0; i < sequenceLength; i++)
947     {
948         links.Delete(sequence[i]);
949     }
950 }
951
952 }
953
954 [Fact(Skip = "Correct implementation is pending")]
955 public static void CalculateAllUsagesTest()
956 {
957     const long sequenceLength = 3;
958
959     using (var scope = new TempLinksTestScope(useSequences: true))
960     {
961         var links = scope.Links;
962         var sequences = scope.Sequences;
963
964         var sequence = new ulong[sequenceLength];
965         for (var i = 0; i < sequenceLength; i++)
966         {
967             sequence[i] = links.Create();
968         }
969
970         var createResults = sequences.CreateAllVariants2(sequence);
971
972         //var reverseResults =
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
983             sequences.CalculateAllUsages2(linksTotalUsages2);
984
985             var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
986             Assert.True(intersection1.Count == linksTotalUsages2.Length);
987         }
988
989         for (var i = 0; i < sequenceLength; i++)
990         {
991             links.Delete(sequence[i]);
992         }
993     }
994 }
995 }

```

1.116 ./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.ResizableDirectMemory.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)

```

```

21     {
22         _deleteFiles = deleteFiles;
23         TempFilename = Path.GetTempFileName();
24         TempTransactionLogFilename = Path.GetTempFileName();
25         var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);
26         MemoryAdapter = useLog ? (ILinks<ulong>)new
            ↳ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
            ↳ coreMemoryAdapter;
27         Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
28         if (useSequences)
29         {
30             Sequences = new Sequences.Sequences(Links, sequencesOptions);
31         }
32     }
33
34     protected override void Dispose(bool manual, bool wasDisposed)
35     {
36         if (!wasDisposed)
37         {
38             Links.Unsync.DisposeIfPossible();
39             if (_deleteFiles)
40             {
41                 DeleteFiles();
42             }
43         }
44     }
45
46     public void DeleteFiles()
47     {
48         File.Delete(TempFilename);
49         File.Delete(TempTransactionLogFilename);
50     }
51 }
52 }

```

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

```

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

```

```

45
46     link = new Link<T>(links.GetLink(linkAddress));
47
48     Assert.True(equalityComparer.Equals(link.Source, linkAddress));
49     Assert.True(equalityComparer.Equals(link.Target, linkAddress));
50
51     // Update link to reference null (prepare for delete)
52     var updated = links.Update(linkAddress, constants.Null, constants.Null);
53
54     Assert.True(equalityComparer.Equals(updated, linkAddress));
55
56     link = new Link<T>(links.GetLink(linkAddress));
57
58     Assert.True(equalityComparer.Equals(link.Source, constants.Null));
59     Assert.True(equalityComparer.Equals(link.Target, constants.Null));
60
61     // Delete link
62     links.Delete(linkAddress);
63
64     Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
65
66     setter = new Setter<T>(constants.Null);
67     links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
68
69     Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
70 }
71
72 public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
73 {
74     // Constants
75     var constants = links.Constants;
76     var equalityComparer = EqualityComparer<T>.Default;
77
78     var h106E = new Hybrid<T>(106L, isExternal: true);
79     var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
80     var h108E = new Hybrid<T>(-108L);
81
82     Assert.Equal(106L, h106E.AbsoluteValue);
83     Assert.Equal(107L, h107E.AbsoluteValue);
84     Assert.Equal(108L, h108E.AbsoluteValue);
85
86     // Create Link (External -> External)
87     var linkAddress1 = links.Create();
88
89     links.Update(linkAddress1, h106E, h108E);
90
91     var link1 = new Link<T>(links.GetLink(linkAddress1));
92
93     Assert.True(equalityComparer.Equals(link1.Source, h106E));
94     Assert.True(equalityComparer.Equals(link1.Target, h108E));
95
96     // Create Link (Internal -> External)
97     var linkAddress2 = links.Create();
98
99     links.Update(linkAddress2, linkAddress1, h108E);
100
101     var link2 = new Link<T>(links.GetLink(linkAddress2));
102
103     Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
104     Assert.True(equalityComparer.Equals(link2.Target, h108E));
105
106     // Create Link (Internal -> Internal)
107     var linkAddress3 = links.Create();
108
109     links.Update(linkAddress3, linkAddress1, linkAddress2);
110
111     var link3 = new Link<T>(links.GetLink(linkAddress3));
112
113     Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
114     Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
115
116     // Search for created link
117     var setter1 = new Setter<T>(constants.Null);
118     links.Each(h106E, h108E, setter1.SetAndReturnFalse);
119
120     Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
121
122     // Search for nonexistent link
123     var setter2 = new Setter<T>(constants.Null);
124     links.Each(h106E, h107E, setter2.SetAndReturnFalse);

```



```

125     Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
126
127     // Update link to reference null (prepare for delete)
128     var updated = links.Update(linkAddress3, constants.Null, constants.Null);
129
130     Assert.True(equalityComparer.Equals(updated, linkAddress3));
131
132     link3 = new Link<T>(links.GetLink(linkAddress3));
133
134     Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
135     Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
136
137     // Delete link
138     links.Delete(linkAddress3);
139
140     Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Two));
141
142     var setter3 = new Setter<T>(constants.Null);
143     links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
144
145     Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
146 }
147
148 public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
149     ↪ links, int maximumOperationsPerCycle)
150 {
151     var comparer = Comparer<TLink>.Default;
152     for (var N = 1; N < maximumOperationsPerCycle; N++)
153     {
154         var random = new System.Random(N);
155         var created = 0;
156         var deleted = 0;
157         for (var i = 0; i < N; i++)
158         {
159             long linksCount = (Integer<TLink>)links.Count();
160             var createPoint = random.NextBoolean();
161             if (linksCount > 2 && createPoint)
162             {
163                 var linksAddressRange = new Range<ulong>(1, (ulong)linksCount);
164                 TLink source = (Integer<TLink>)random.NextUInt64(linksAddressRange);
165                 TLink target = (Integer<TLink>)random.NextUInt64(linksAddressRange);
166                 ↪ // -V3086
167                 var resultLink = links.CreateAndUpdate(source, target);
168                 if (comparer.Compare(resultLink, (Integer<TLink>)linksCount) > 0)
169                 {
170                     created++;
171                 }
172             }
173             else
174             {
175                 links.Create();
176                 created++;
177             }
178         }
179         Assert.True(created == (Integer<TLink>)links.Count());
180         for (var i = 0; i < N; i++)
181         {
182             TLink link = (Integer<TLink>)(i + 1);
183             if (links.Exists(link))
184             {
185                 links.Delete(link);
186                 deleted++;
187             }
188         }
189         Assert.True((Integer<TLink>)links.Count() == 0);
190     }
191 }
192 }

```

1.118 ./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.ResizableDirectMemory.Specific;
22
23 namespace Platform.Data.Doublets.Tests
24 {
25     public static class UInt64LinksTests
26     {
27         private static readonly LinksConstants<ulong> _constants =
28             ↪ Default<LinksConstants<ulong>>.Instance;
29
30         private const long Iterations = 10 * 1024;
31
32         #region Concept
33
34         [Fact]
35         public static void MultipleCreateAndDeleteTest()
36         {
37             using (var scope = new Scope<Types<HeapResizableDirectMemory,
38                 ↪ UInt64ResizableDirectMemoryLinks>>())
39             {
40                 new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeletions(100);
41             }
42
43             [Fact]
44             public static void CascadeUpdateTest()
45             {
46                 var itself = _constants.Itself;
47                 using (var scope = new TempLinksTestScope(useLog: true))
48                 {
49                     var links = scope.Links;
50
51                     var l1 = links.Create();
52                     var l2 = links.Create();
53
54                     l2 = links.Update(l2, l2, l1, l2);
55
56                     links.CreateAndUpdate(l2, itself);
57                     links.CreateAndUpdate(l2, itself);
58
59                     l2 = links.Update(l2, l1);
60
61                     links.Delete(l2);
62
63                     Global.Trash = links.Count();
64
65                     links.Unsync.DisposeIfPossible(); // Close links to access log
66
67                     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope,
68                         ↪ e.TempTransactionLogFilename);
69                 }
70
71                 [Fact]
72                 public static void BasicTransactionLogTest()
73                 {
74                     using (var scope = new TempLinksTestScope(useLog: true))
75                     {
76                         var links = scope.Links;
77                         var l1 = links.Create();
78                         var l2 = links.Create();
79
80                         Global.Trash = links.Update(l2, l2, l1, l2);
81
82                         links.Delete(l1);
83
84                         links.Unsync.DisposeIfPossible(); // Close links to access log

```

```

85         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope ↵
86         ↵ e.TempTransactionLogFilename);
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(0UL, links.Count());
106
107         links.Unsync.DisposeIfPossible();
108
109         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s ↵
110         ↵ cope.TempTransactionLogFilename);
111         Assert.Single(transitions);
112     }
113 }
114 [Fact]
115 public static void TransactionUserCodeErrorNoDataSavedTest()
116 {
117     // User Code Error (Autoreverted), no data saved
118     var itself = _constants.Itself;
119
120     TempLinksTestScope lastScope = null;
121     try
122     {
123         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false, ↵
124         ↵ useLog: true))
125         {
126             var links = scope.Links;
127             var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor ↵
128             ↵ atorBase<ulong>)links.Unsync).Links;
129             using (var transaction = transactionsLayer.BeginTransaction())
130             {
131                 var l1 = links.CreateAndUpdate(itself, itself);
132                 var l2 = links.CreateAndUpdate(itself, itself);
133
134                 l2 = links.Update(l2, l2, l1, l2);
135
136                 links.CreateAndUpdate(l2, itself);
137                 links.CreateAndUpdate(l2, itself);
138
139                 //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi ↵
140                 ↵ tion>(scope.TempTransactionLogFilename);
141
142                 l2 = links.Update(l2, l1);
143
144                 links.Delete(l2);
145
146                 ExceptionThrower();
147
148                 transaction.Commit();
149             }
150
151             Global.Trash = links.Count();
152         }
153     }
154     catch
155     {
156         Assert.False(lastScope == null);
157
158         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l ↵
159         ↵ astScope.TempTransactionLogFilename);
160     }

```

```

157         Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
158             ↳ transitions[0].After.IsNull());
159
160         lastScope.DeleteFiles();
161     }
162 }
163 [Fact]
164 public static void TransactionUserCodeErrorSomeDataSavedTest()
165 {
166     // User Code Error (Autoreverted), some data saved
167     var itself = _constants.Itself;
168
169     TempLinksTestScope lastScope = null;
170     try
171     {
172         ulong l1;
173         ulong l2;
174
175         using (var scope = new TempLinksTestScope(useLog: true))
176         {
177             var links = scope.Links;
178             l1 = links.CreateAndUpdate(itself, itself);
179             l2 = links.CreateAndUpdate(itself, itself);
180
181             l2 = links.Update(l2, l2, l1, l2);
182
183             links.CreateAndUpdate(l2, itself);
184             links.CreateAndUpdate(l2, itself);
185
186             links.Unsync.DisposeIfPossible();
187
188             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
189                 ↳ scope.TempTransactionLogFilename);
190         }
191
192         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
193             ↳ useLog: true))
194         {
195             var links = scope.Links;
196             var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
197             using (var transaction = transactionsLayer.BeginTransaction())
198             {
199                 l2 = links.Update(l2, l1);
200
201                 links.Delete(l2);
202
203                 ExceptionThrower();
204
205                 transaction.Commit();
206             }
207
208             Global.Trash = links.Count();
209         }
210     }
211     catch
212     {
213         Assert.False(lastScope == null);
214
215         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
216             ↳ Scope.TempTransactionLogFilename);
217
218         lastScope.DeleteFiles();
219     }
220 }
221 [Fact]
222 public static void TransactionCommit()
223 {
224     var itself = _constants.Itself;
225
226     var tempDatabaseFilename = Path.GetTempFileName();
227     var tempTransactionLogFilename = Path.GetTempFileName();
228
229     // Commit
230     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
231         ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
232         ↳ tempTransactionLogFilename))
233     using (var links = new UInt64Links(memoryAdapter))

```

```

230     {
231         using (var transaction = memoryAdapter.BeginTransaction())
232         {
233             var l1 = links.CreateAndUpdate(itself, itself);
234             var l2 = links.CreateAndUpdate(itself, itself);
235
236             Global.Trash = links.Update(l2, l2, l1, l2);
237
238             links.Delete(l1);
239
240             transaction.Commit();
241         }
242
243         Global.Trash = links.Count();
244     }
245
246     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
    ↪ sactionLogFilename);
247 }
248
249 [Fact]
250 public static void TransactionDamage()
251 {
252     var itself = _constants.Itself;
253
254     var tempDatabaseFilename = Path.GetTempFileName();
255     var tempTransactionLogFilename = Path.GetTempFileName();
256
257     // Commit
258     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
    ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
    ↪ tempTransactionLogFilename))
259     using (var links = new UInt64Links(memoryAdapter))
260     {
261         using (var transaction = memoryAdapter.BeginTransaction())
262         {
263             var l1 = links.CreateAndUpdate(itself, itself);
264             var l2 = links.CreateAndUpdate(itself, itself);
265
266             Global.Trash = links.Update(l2, l2, l1, l2);
267
268             links.Delete(l1);
269
270             transaction.Commit();
271         }
272
273         Global.Trash = links.Count();
274     }
275
276     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
    ↪ sactionLogFilename);
277
278     // Damage database
279
280     FileHelpers.WriteFirst(tempTransactionLogFilename, new
    ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
281
282     // Try load damaged database
283     try
284     {
285         // TODO: Fix
286         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
    ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
    ↪ tempTransactionLogFilename))
287         using (var links = new UInt64Links(memoryAdapter))
288         {
289             Global.Trash = links.Count();
290         }
291     }
292     catch (NotSupportedException ex)
293     {
294         Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
    ↪ yet.");
295     }
296
297     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
    ↪ sactionLogFilename);
298
299     File.Delete(tempDatabaseFilename);

```

```

300     File.Delete(tempTransactionLogFilename);
301 }
302
303 [Fact]
304 public static void Bug1Test()
305 {
306     var tempDatabaseFilename = Path.GetTempFileName();
307     var tempTransactionLogFilename = Path.GetTempFileName();
308
309     var itself = _constants.Itself;
310
311     // User Code Error (Autoreverted), some data saved
312     try
313     {
314         ulong l1;
315         ulong l2;
316
317         using (var memory = new UInt64ResizableDirectMemoryLinks(tempDatabaseFilename))
318         using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
319             ↪ tempTransactionLogFilename))
320         using (var links = new UInt64Links(memoryAdapter))
321         {
322             l1 = links.CreateAndUpdate(itself, itself);
323             l2 = links.CreateAndUpdate(itself, itself);
324
325             l2 = links.Update(l2, l2, l1, l2);
326
327             links.CreateAndUpdate(l2, itself);
328             links.CreateAndUpdate(l2, itself);
329         }
330
331         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
332             ↪ TransactionLogFilename);
333
334         using (var memory = new UInt64ResizableDirectMemoryLinks(tempDatabaseFilename))
335         using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
336             ↪ tempTransactionLogFilename))
337         using (var links = new UInt64Links(memoryAdapter))
338         {
339             using (var transaction = memoryAdapter.BeginTransaction())
340             {
341                 l2 = links.Update(l2, l1);
342
343                 links.Delete(l2);
344
345                 ExceptionThrower();
346
347                 transaction.Commit();
348             }
349
350             Global.Trash = links.Count();
351         }
352     }
353     catch
354     {
355         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
356             ↪ TransactionLogFilename);
357     }
358
359     File.Delete(tempDatabaseFilename);
360     File.Delete(tempTransactionLogFilename);
361 }
362
363 private static void ExceptionThrower() => throw new InvalidOperationException();
364
365 [Fact]
366 public static void PathsTest()
367 {
368     var source = _constants.SourcePart;
369     var target = _constants.TargetPart;
370
371     using (var scope = new TempLinksTestScope())
372     {
373         var links = scope.Links;
374         var l1 = links.CreatePoint();
375         var l2 = links.CreatePoint();
376
377         var r1 = links.GetByKeys(l1, source, target, source);
378         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
379     }
380 }

```

```

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.CreateAndUpdate(a, b);
391         var cb = links.CreateAndUpdate(c, b);
392         var ac = links.CreateAndUpdate(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) ==
403             ↪ "(5:(4:5 (6:5 4)) 6)");
404         Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
405             ↪ "(6:(5:(4:5 6) 6) 4)");
406         Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
407             ↪ "(4:(5:4 (6:5 4)) 6)");
408
409         // TODO: Think how to build balanced syntax tree while formatting structure (eg.
410         ↪ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)"
411
412         Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
413             ↪ "{{5}{5}{4}{6}}");
414         Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
415             ↪ "{{5}{6}{6}{4}}");
416         Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
417             ↪ "{{4}{5}{4}{6}}");
418     }
419 }
420
421 private static void DefaultFormatter(StringBuilder sb, ulong link)
422 {
423     sb.Append(link.ToString());
424 }
425
426 #endregion
427
428 #region Performance
429
430 /*
431 public static void RunAllPerformanceTests()
432 {
433     try
434     {
435         links.TestLinksInSteps();
436     }
437     catch (Exception ex)
438     {
439         ex.WriteToConsole();
440     }
441
442     return;
443
444     try
445     {
446         //ThreadPool.SetMaxThreads(2, 2);
447
448         // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
449         ↪ результат
450         // Также это дополнительно помогает в отладке
451         // Увеличивает вероятность попадания информации в кэши
452         for (var i = 0; i < 10; i++)
453         {

```

```

446         //0 - 10 ГБ
447         //Каждые 100 МБ срез цифр
448
449         //links.TestGetSourceFunction();
450         //links.TestGetSourceFunctionInParallel();
451         //links.TestGetTargetFunction();
452         //links.TestGetTargetFunctionInParallel();
453         links.Create64BillionLinks();
454
455         links.TestRandomSearchFixed();
456         //links.Create64BillionLinksInParallel();
457         links.TestEachFunction();
458         //links.TestForeach();
459         //links.TestParallelForeach();
460     }
461
462     links.TestDeletionOfAllLinks();
463
464 }
465 catch (Exception ex)
466 {
467     ex.WriteToConsole();
468 }
469 }*/
470
471 /*
472 public static void TestLinksInSteps()
473 {
474     const long gibibyte = 1024 * 1024 * 1024;
475     const long mebibyte = 1024 * 1024;
476
477     var totalLinksToCreate = gibibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
478     var linksStep = 102 * mebibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
479
480     var creationMeasurements = new List<TimeSpan>();
481     var searchMeasurements = new List<TimeSpan>();
482     var deletionMeasurements = new List<TimeSpan>();
483
484     GetBaseRandomLoopOverhead(linksStep);
485     GetBaseRandomLoopOverhead(linksStep);
486
487     var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
488
489     ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
490
491     var loops = totalLinksToCreate / linksStep;
492
493     for (int i = 0; i < loops; i++)
494     {
495         creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
496         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}",
584                 Iterations, elapsedTime, (long)iterationsPerSecond, counter);
585         }
586     }
587
588     [Fact(Skip = "performance test")]
589     public static void GetSourceInParallel()
590     {
591         using (var scope = new TempLinksTestScope())
592         {
593             var links = scope.Links;
594             ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
↵ parallel.", Iterations);

```

```

595     long counter = 0;
596
597     //var firstLink = links.First();
598     var firstLink = links.Create();
599
600
601     var sw = Stopwatch.StartNew();
602
603     // Тестируем саму функцию
604     Parallel.For(0, Iterations, x =>
605     {
606         Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
607         //Interlocked.Increment(ref counter);
608     });
609
610     var elapsedTime = sw.Elapsed;
611
612     var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
613
614     links.Delete(firstLink);
615
616     ConsoleHelpers.Debug(
617         "{0} Iterations of GetSource function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
618     }
619 }
620
621 [Fact(Skip = "performance test")]
622 public static void TestGetTarget()
623 {
624     using (var scope = new TempLinksTestScope())
625     {
626         var links = scope.Links;
627         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
628             ↳ Iterations);
629
630         ulong counter = 0;
631
632         //var firstLink = links.First();
633         var firstLink = links.Create();
634
635         var sw = Stopwatch.StartNew();
636
637         for (ulong i = 0; i < Iterations; i++)
638         {
639             counter += links.GetTarget(firstLink);
640         }
641
642         var elapsedTime = sw.Elapsed;
643
644         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
645
646         links.Delete(firstLink);
647
648         ConsoleHelpers.Debug(
649             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
650     }
651 }
652
653 [Fact(Skip = "performance test")]
654 public static void TestGetTargetInParallel()
655 {
656     using (var scope = new TempLinksTestScope())
657     {
658         var links = scope.Links;
659         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
660             ↳ 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 // TODO: Заполнить базу данных перед тестом
687 /*
688 [Fact]
689 public void TestRandomSearchFixed()
690 {
691     var tempFilename = Path.GetTempFileName();
692
693     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
694 ↪ DefaultLinksSizeStep))
695     {
696         long iterations = 64 * 1024 * 1024 /
697 ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
698
699         ulong counter = 0;
700         var maxLink = links.Total;
701
702         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
703
704         var sw = Stopwatch.StartNew();
705
706         for (var i = iterations; i > 0; i--)
707         {
708             var source =
709 ↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
710             var target =
711 ↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
712
713             counter += links.Search(source, target);
714         }
715
716         var elapsedTime = sw.Elapsed;
717
718         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
719
720         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
721 ↪ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
722 ↪ counter);
723     }
724
725     File.Delete(tempFilename);
726 }*/
727
728 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
729 public static void TestRandomSearchAll()
730 {
731     using (var scope = new TempLinksTestScope())
732     {
733         var links = scope.Links;
734         ulong counter = 0;
735
736         var maxLink = links.Count();
737
738         var iterations = links.Count();
739
740         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
741 ↪ links.Count());
742
743         var sw = Stopwatch.StartNew();
744
745         for (var i = iterations; i > 0; i--)
746         {
747             var linksAddressRange = new
748 ↪ Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);

```

```

742         var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
743         var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
744
745         counter += links.SearchOrDefault(source, target);
746     }
747
748     var elapsedTime = sw.Elapsed;
749
750     var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
751
752     ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
753     ↪     Iterations per second), c: {3}",
754         iterations, elapsedTime, (long)iterationsPerSecond, counter);
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}
777         ↪     links per second)",
778             counter, elapsedTime, (long)linksPerSecond);
779     }
780 }
781
782 /*
783 [Fact]
784 public static void TestForeach()
785 {
786     var tempFilename = Path.GetTempFileName();
787
788     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
789     ↪     DefaultLinksSizeStep))
790     {
791         ulong counter = 0;
792
793         ConsoleHelpers.Debug("Testing foreach through links.");
794
795         var sw = Stopwatch.StartNew();
796
797         //foreach (var link in links)
798         //{
799             //    counter++;
800         //}
801
802         var elapsedTime = sw.Elapsed;
803
804         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
805
806         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
807         ↪     links per second)", counter, elapsedTime, (long)linksPerSecond);
808     }
809
810     File.Delete(tempFilename);
811 }
812 */
813
814 /*
815 [Fact]
816 public static void TestParallelForeach()
817 {
818     var tempFilename = Path.GetTempFileName();

```

```

818         using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪ DefaultLinksSizeStep))
819         {
820             long counter = 0;
821
822             ConsoleHelpers.Debug("Testing parallel foreach through links.");
823
824             var sw = Stopwatch.StartNew();
825
826             //Parallel.ForEach((IEnumerable<ulong>)links, x =>
827             //{
828             //    Interlocked.Increment(ref counter);
829             //});
830
831             var elapsedTime = sw.Elapsed;
832
833             var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
834
835             ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↪ {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
836         }
837
838         File.Delete(tempFilename);
839     }
840     */
841
842     [Fact(Skip = "performance test")]
843     public static void Create64BillionLinks()
844     {
845         using (var scope = new TempLinksTestScope())
846         {
847             var links = scope.Links;
848             var linksBeforeTest = links.Count();
849
850             long linksToCreate = 64 * 1024 * 1024 /
↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
851
852             ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
853
854             var elapsedTime = Performance.Measure(() =>
855             {
856                 for (long i = 0; i < linksToCreate; i++)
857                 {
858                     links.Create();
859                 }
860             });
861
862             var linksCreated = links.Count() - linksBeforeTest;
863             var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
864
865             ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
866
867             ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
↪ linksCreated, elapsedTime,
868                 (long)linksPerSecond);
869         }
870     }
871
872     [Fact(Skip = "performance test")]
873     public static void Create64BillionLinksInParallel()
874     {
875         using (var scope = new TempLinksTestScope())
876         {
877             var links = scope.Links;
878             var linksBeforeTest = links.Count();
879
880             var sw = Stopwatch.StartNew();
881
882             long linksToCreate = 64 * 1024 * 1024 /
↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
883
884             ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
885
886             Parallel.For(0, linksToCreate, x => links.Create());
887
888             var elapsedTime = sw.Elapsed;
889
890             var linksCreated = links.Count() - linksBeforeTest;
891             var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
892

```

```

893     ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
894         ↪ linksCreated, elapsedTime,
895         (long)linksPerSecond);
896     }
897 }
898
899 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
900 public static void TestDeletionOfAllLinks()
901 {
902     using (var scope = new TempLinksTestScope())
903     {
904         var links = scope.Links;
905         var linksBeforeTest = links.Count();
906
907         ConsoleHelpers.Debug("Deleting all links");
908
909         var elapsedTime = Performance.Measure(links.DeleteAll);
910
911         var linksDeleted = linksBeforeTest - links.Count();
912         var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
913
914         ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
915             ↪ linksDeleted, elapsedTime,
916             (long)linksPerSecond);
917     }
918 }
919 #endregion
920 }
921 }

```

1.119 ./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     }
45 }

```

1.120 ./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.ResizableDirectMemory.Generic;
15
16 namespace Platform.Data.Doublets.Tests
17 {
18     public static class UnicodeConvertersTests
19     {
20         [Fact]
21         public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
22         {
23             using (var scope = new TempLinksTestScope())
24             {
25                 var links = scope.Links;
26                 var meaningRoot = links.CreatePoint();
27                 var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
28                 var powerOf2ToUnaryNumberConverter = new
29                     ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
30                 var addressToUnaryNumberConverter = new
31                     ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
32                 var unaryNumberToAddressConverter = new
33                     ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
34                     ↪ powerOf2ToUnaryNumberConverter);
35                 TestCharAndUnicodeSymbolConverters(links, meaningRoot,
36                     ↪ addressToUnaryNumberConverter, unaryNumberToAddressConverter);
37             }
38         }
39
40         [Fact]
41         public static void CharAndRawNumberUnicodeSymbolConvertersTest()
42         {
43             using (var scope = new Scope<Types<HeapResizableDirectMemory,
44                 ↪ ResizableDirectMemoryLinks<ulong>>>())
45             {
46                 var links = scope.Use<ILinks<ulong>>>();
47                 var meaningRoot = links.CreatePoint();
48                 var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
49                 var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
50                 TestCharAndUnicodeSymbolConverters(links, meaningRoot,
51                     ↪ addressToRawNumberConverter, rawNumberToAddressConverter);
52             }
53         }
54
55         private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
56             ↪ meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
57             ↪ numberToAddressConverter)
58         {
59             var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
60             var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
61                 ↪ addressToNumberConverter, unicodeSymbolMarker);
62             var originalCharacter = 'H';
63             var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
64             var unicodeSymbolCriterionMatcher = new UnicodeSymbolCriterionMatcher<ulong>(links,
65                 ↪ unicodeSymbolMarker);
66             var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
67                 ↪ numberToAddressConverter, unicodeSymbolCriterionMatcher);
68             var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
69             Assert.Equal(originalCharacter, resultingCharacter);
70         }
71
72         [Fact]
73         public static void StringAndUnicodeSequenceConvertersTest()
74         {
75             using (var scope = new TempLinksTestScope())
76             {
77                 var links = scope.Links;

```

```

67     var itself = links.Constants.Itself;
68
69     var meaningRoot = links.CreatePoint();
70     var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
71     var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
72     var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
73     var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
74     var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
75
76     var powerOf2ToUnaryNumberConverter = new
77     ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
78     var addressToUnaryNumberConverter = new
79     ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
80     var charToUnicodeSymbolConverter = new
81     ↪ CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
82     ↪ unicodeSymbolMarker);
83
84     var unaryNumberToAddressConverter = new
85     ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
86     ↪ powerOf2ToUnaryNumberConverter);
87     var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
88     var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
89     ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
90     var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
91     ↪ frequencyPropertyMarker, frequencyMarker);
92     var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
93     ↪ frequencyPropertyOperator, frequencyIncrementer);
94     var linkToItsFrequencyNumberConverter = new
95     ↪ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
96     ↪ unaryNumberToAddressConverter);
97     var sequenceToItsLocalElementLevelsConverter = new
98     ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
99     ↪ linkToItsFrequencyNumberConverter);
100    var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
101    ↪ sequenceToItsLocalElementLevelsConverter);
102
103    var stringToUnicodeSequenceConverter = new
104    ↪ StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
105    ↪ index, optimalVariantConverter, unicodeSequenceMarker);
106
107    var originalString = "Hello";
108
109    var unicodeSequenceLink =
110    ↪ stringToUnicodeSequenceConverter.Convert(originalString);
111
112    var unicodeSymbolCriterionMatcher = new
113    ↪ UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
114    var unicodeSymbolToCharConverter = new
115    ↪ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
116    ↪ unicodeSymbolCriterionMatcher);
117
118    var unicodeSequenceCriterionMatcher = new
119    ↪ UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
120
121    var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
122    ↪ unicodeSymbolCriterionMatcher.IsMatched);
123
124    var unicodeSequenceToStringConverter = new
125    ↪ UnicodeSequenceToStringConverter<ulong>(links,
126    ↪ unicodeSequenceCriterionMatcher, sequenceWalker,
127    ↪ unicodeSymbolToCharConverter);
128
129    var resultingString =
130    ↪ unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
131
132    Assert.Equal(originalString, resultingString);
133
134    }
135
136    }
137
138    }
139
140    }

```


Index

- ./Platform.Data.Doublets.Tests/ComparisonTests.cs, 143
- ./Platform.Data.Doublets.Tests/EqualityTests.cs, 144
- ./Platform.Data.Doublets.Tests/GenericLinksTests.cs, 145
- ./Platform.Data.Doublets.Tests/LinksConstantsTests.cs, 146
- ./Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs, 146
- ./Platform.Data.Doublets.Tests/ReadSequenceTests.cs, 149
- ./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs, 150
- ./Platform.Data.Doublets.Tests/ScopeTests.cs, 151
- ./Platform.Data.Doublets.Tests/SequencesTests.cs, 152
- ./Platform.Data.Doublets.Tests/TempLinksTestScope.cs, 166
- ./Platform.Data.Doublets.Tests/TestExtensions.cs, 167
- ./Platform.Data.Doublets.Tests/UInt64LinksTests.cs, 169
- ./Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs, 182
- ./Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs, 182
- ./Platform.Data.Doublets/Decorators/LinksCascadeUniquenessAndUsagesResolver.cs, 1
- ./Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.cs, 1
- ./Platform.Data.Doublets/Decorators/LinksDecoratorBase.cs, 1
- ./Platform.Data.Doublets/Decorators/LinksDisposableDecoratorBase.cs, 2
- ./Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.cs, 3
- ./Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.cs, 3
- ./Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.cs, 4
- ./Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.cs, 4
- ./Platform.Data.Doublets/Decorators/LinksUniquenessResolver.cs, 5
- ./Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs, 5
- ./Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs, 5
- ./Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs, 6
- ./Platform.Data.Doublets/Decorators/UInt64Links.cs, 6
- ./Platform.Data.Doublets/Decorators/UniLinks.cs, 7
- ./Platform.Data.Doublets/Doublet.cs, 12
- ./Platform.Data.Doublets/DoubletComparer.cs, 12
- ./Platform.Data.Doublets/Hybrid.cs, 13
- ./Platform.Data.Doublets/ILinks.cs, 14
- ./Platform.Data.Doublets/ILinksExtensions.cs, 15
- ./Platform.Data.Doublets/ISynchronizedLinks.cs, 26
- ./Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs, 26
- ./Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs, 26
- ./Platform.Data.Doublets/Link.cs, 27
- ./Platform.Data.Doublets/LinkExtensions.cs, 30
- ./Platform.Data.Doublets/LinksOperatorBase.cs, 30
- ./Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs, 30
- ./Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConverter.cs, 31
- ./Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs, 31
- ./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs, 32
- ./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs, 33
- ./Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 33
- ./Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 34
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksAvlBalancedTreeMethodsBase.cs, 35
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSizeBalancedTreeMethodsBase.cs, 39
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesAvlBalancedTreeMethods.cs, 42
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesSizeBalancedTreeMethods.cs, 43
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsAvlBalancedTreeMethods.cs, 44
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsSizeBalancedTreeMethods.cs, 45
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinks.cs, 46
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinksBase.cs, 47
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/UnusedLinksListMethods.cs, 54
- ./Platform.Data.Doublets/ResizableDirectMemory/ILinksListMethods.cs, 55
- ./Platform.Data.Doublets/ResizableDirectMemory/ILinksTreeMethods.cs, 55
- ./Platform.Data.Doublets/ResizableDirectMemory/LinksHeader.cs, 55
- ./Platform.Data.Doublets/ResizableDirectMemory/RawLink.cs, 56
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs, 56
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs, 58
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs, 59
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.cs, 60
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs, 61
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs, 62
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64ResizableDirectMemoryLinks.cs, 63

./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64UnusedLinksListMethods.cs, 65
./Platform.Data.Doublets/Sequences/ArrayExtensions.cs, 65
./Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 66
./Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 66
./Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 69
./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 70
./Platform.Data.Doublets/Sequences/Converters/SequenceToToltsLocalElementLevelsConverter.cs, 71
./Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs, 72
./Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs, 72
./Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 72
./Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 73
./Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 73
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 75
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 77
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToltsFrequencyValueConverter.cs, 77
./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 78
./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 78
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 79
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 79
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 79
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 80
./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 81
./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 81
./Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 82
./Platform.Data.Doublets/Sequences/IListExtensions.cs, 82
./Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 82
./Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 83
./Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 84
./Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 84
./Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 85
./Platform.Data.Doublets/Sequences/Indexes/Unindex.cs, 85
./Platform.Data.Doublets/Sequences/ListFiller.cs, 85
./Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 86
./Platform.Data.Doublets/Sequences/Sequences.cs, 112
./Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 123
./Platform.Data.Doublets/Sequences/SequencesOptions.cs, 123
./Platform.Data.Doublets/Sequences/SetFiller.cs, 125
./Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 125
./Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 126
./Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 126
./Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 128
./Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 128
./Platform.Data.Doublets/Stacks/Stack.cs, 129
./Platform.Data.Doublets/Stacks/StackExtensions.cs, 130
./Platform.Data.Doublets/SynchronizedLinks.cs, 130
./Platform.Data.Doublets/UInt64LinksExtensions.cs, 131
./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 133
./Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 138
./Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 138
./Platform.Data.Doublets/Unicode/UnicodeMap.cs, 139
./Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs, 141
./Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 141
./Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs, 142
./Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 142