

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     {
23         ↪ Links.Update(restrictions, Links.ResolveConstantAsSelfReference(Constants.Null,
24             ↪ restrictions, substitution));
25     }
26 }

```

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

```

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

```

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

```

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

```

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

```

```

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/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 }

```

1.18 ./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.GetOrCreate(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             }
59         }
60     }
61 }

```

```

55     }
56
57     public static void Delete<TLink>(this ILinks<TLink> links, TLink linkToDelete) =>
58         ↪ links.Delete(new LinkAddress<TLink>(linkToDelete));
59
60     /// <remarks>
61     /// TODO: Возможно есть очень простой способ это сделать.
62     /// (Например просто удалить файл, или изменить его размер таким образом,
63     /// чтобы удалился весь контент)
64     /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
65     /// </remarks>
66     public static void DeleteAll<TLink>(this ILinks<TLink> links)
67     {
68         var equalityComparer = EqualityComparer<TLink>.Default;
69         var comparer = Comparer<TLink>.Default;
70         for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
71             ↪ Arithmetic.Decrement(i))
72         {
73             links.Delete(i);
74             if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
75             {
76                 i = links.Count();
77             }
78         }
79     }
80
81     public static TLink First<TLink>(this ILinks<TLink> links)
82     {
83         TLink firstLink = default;
84         var equalityComparer = EqualityComparer<TLink>.Default;
85         if (equalityComparer.Equals(links.Count(), default))
86         {
87             throw new InvalidOperationException("В хранилище нет связей.");
88         }
89         links.Each(links.Constants.Any, links.Constants.Any, link =>
90         {
91             firstLink = link[links.Constants.IndexPart];
92             return links.Constants.Break;
93         });
94         if (equalityComparer.Equals(firstLink, default))
95         {
96             throw new InvalidOperationException("В процессе поиска по хранилищу не было
97                 ↪ найдено связей.");
98         }
99         return firstLink;
100     }
101
102     #region Paths
103
104     /// <remarks>
105     /// TODO: Как так? Как то что ниже может быть корректно?
106     /// Скорее всего практически не применимо
107     /// Предполагалось, что можно было конвертировать формируемый в проходе через
108     ↪ SequenceWalker
109     /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
110     /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
111     /// </remarks>
112     public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
113         ↪ path)
114     {
115         var current = path[0];
116         //EnsureLinkExists(current, "path");
117         if (!links.Exists(current))
118         {
119             return false;
120         }
121         var equalityComparer = EqualityComparer<TLink>.Default;
122         var constants = links.Constants;
123         for (var i = 1; i < path.Length; i++)
124         {
125             var next = path[i];
126             var values = links.GetLink(current);
127             var source = values[constants.SourcePart];
128             var target = values[constants.TargetPart];
129             if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
130                 ↪ next))
131             {
132                 //throw new InvalidOperationException(string.Format("Невозможно выбрать
133                     ↪ 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,
130         ↪ target))
131     {
132         //throw new InvalidOperationException(string.Format("Невозможно продолжить
133         ↪ путь через элемент пути {0}", next));
134         return false;
135     }
136     current = next;
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>

```

```

196 /// <param name="links">Хранилище связей.</param>
197 /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↳ содержимого.</param>
198 /// <returns>Индекс начальной связи для указанной связи.</returns>
199 [MethodImpl(MethodImplOptions.AggressiveInlining)]
200 public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↳ link[links.Constants.SourcePart];

201
202 /// <summary>
203 /// Возвращает индекс конечной (Target) связи для указанной связи.
204 /// </summary>
205 /// <param name="links">Хранилище связей.</param>
206 /// <param name="link">Индекс связи.</param>
207 /// <returns>Индекс конечной связи для указанной связи.</returns>
208 [MethodImpl(MethodImplOptions.AggressiveInlining)]
209 public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
    ↳ links.GetLink(link)[links.Constants.TargetPart];

210
211 /// <summary>
212 /// Возвращает индекс конечной (Target) связи для указанной связи.
213 /// </summary>
214 /// <param name="links">Хранилище связей.</param>
215 /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↳ содержимого.</param>
216 /// <returns>Индекс конечной связи для указанной связи.</returns>
217 [MethodImpl(MethodImplOptions.AggressiveInlining)]
218 public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↳ link[links.Constants.TargetPart];

219
220 /// <summary>
221 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
222 /// </summary>
223 /// <param name="links">Хранилище связей.</param>
224 /// <param name="handler">Обработчик каждой подходящей связи.</param>
225 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
    ↳ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
    ↳ Any - отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
226 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
227 [MethodImpl(MethodImplOptions.AggressiveInlining)]
228 public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
    ↳ handler, params TLink[] restrictions)
229 => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
    ↳ links.Constants.Continue);

230
231 /// <summary>
232 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
233 /// </summary>
234 /// <param name="links">Хранилище связей.</param>
235 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
    ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
236 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
    ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
237 /// <param name="handler">Обработчик каждой подходящей связи.</param>
238 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
239 [MethodImpl(MethodImplOptions.AggressiveInlining)]
240 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
    ↳ Func<TLink, bool> handler)
241 {
242     var constants = links.Constants;
243     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
    ↳ constants.Break, constants.Any, source, target);
244 }
245
246 /// <summary>
247 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
248 /// </summary>
249 /// <param name="links">Хранилище связей.</param>

```



```

250 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
251   ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
252     Constants.Any - любое начало, 1..∞ конкретное начало)</param>
253 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
254   ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
255     Constants.Any - любой конец, 1..∞ конкретный конец)</param>
256 /// <param name="handler">Обработчик каждой подходящей связи.</param>
257 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
258   ↳ случае.</returns>
259 [MethodImpl(MethodImplOptions.AggressiveInlining)]
260 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
261   ↳ Func<IList<TLink>, TLink> handler)
262 {
263     var constants = links.Constants;
264     return links.Each(handler, constants.Any, source, target);
265 }
266
267 [MethodImpl(MethodImplOptions.AggressiveInlining)]
268 public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
269   ↳ restrictions)
270 {
271     long arraySize = (Integer<TLink>)links.Count(restrictions);
272     var array = new IList<TLink>[arraySize];
273     if (arraySize > 0)
274     {
275         var filler = new ArrayFiller<IList<TLink>, TLink>(array,
276           ↳ links.Constants.Continue);
277         links.Each(filler.AddAndReturnConstant, restrictions);
278     }
279     return array;
280 }
281
282 [MethodImpl(MethodImplOptions.AggressiveInlining)]
283 public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
284   ↳ restrictions)
285 {
286     long arraySize = (Integer<TLink>)links.Count(restrictions);
287     var array = new TLink[arraySize];
288     if (arraySize > 0)
289     {
290         var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
291         links.Each(filler.AddFirstAndReturnConstant, restrictions);
292     }
293     return array;
294 }
295
296 /// <summary>
297 /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
298   ↳ в хранилище связей.
299 /// </summary>
300 /// <param name="links">Хранилище связей.</param>
301 /// <param name="source">Начало связи.</param>
302 /// <param name="target">Конец связи.</param>
303 /// <returns>Значение, определяющее существует ли связь.</returns>
304 [MethodImpl(MethodImplOptions.AggressiveInlining)]
305 public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
306   ↳ => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
307   ↳ default) > 0;
308
309 #region Ensure
310 // TODO: May be move to EnsureExtensions or make it both there and here
311
312 [MethodImpl(MethodImplOptions.AggressiveInlining)]
313 public static void EnsureLinkExists<TLink>(this ILinks<TLink> links, IList<TLink>
314   ↳ restrictions)
315 {
316     for (var i = 0; i < restrictions.Count; i++)
317     {
318         if (!links.Exists(restrictions[i]))
319         {
320             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
321               ↳ $"sequence[{i}]");
322         }
323     }
324 }
325
326 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

```

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

```

```

451 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
452 /// </summary>
453 /// <param name="links">Хранилище связей.</param>
454 /// <param name="source">Индекс связи, которая является началом для искомой
    → связи.</param>
455 /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
456 /// <returns>Индекс искомой связи с указанными Source (началом) и Target
    → (концом).</returns>
457 [MethodImpl(MethodImplOptions.AggressiveInlining)]
458 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
    → target)
459 {
460     var 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
    ↳ с указанными Source (началом) и Target (концом).
538 /// </summary>
539 /// <param name="links">Хранилище связей.</param>
540 /// <param name="source">Индекс связи, которая является началом на создаваемой
    ↳ связи.</param>
541 /// <param name="target">Индекс связи, которая является концом для создаваемой
    ↳ связи.</param>
542 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
543 [MethodImpl(MethodImplOptions.AggressiveInlining)]
544 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target)
545 {
546     var link = links.SearchOrDefault(source, target);
547     if (EqualityComparer<TLink>.Default.Equals(link, default))
548     {
549         link = links.CreateAndUpdate(source, target);
550     }
551     return link;
552 }
553
554 /// <summary>
555 /// Обновляет связь с указанными началом (Source) и концом (Target)
556 /// на связь с указанными началом (NewSource) и концом (NewTarget).
557 /// </summary>
558 /// <param name="links">Хранилище связей.</param>
559 /// <param name="source">Индекс связи, которая является началом обновляемой
    ↳ связи.</param>
560 /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
561 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    ↳ выполняется обновление.</param>
562 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    ↳ выполняется обновление.</param>
563 /// <returns>Индекс обновлённой связи.</returns>
564 [MethodImpl(MethodImplOptions.AggressiveInlining)]
565 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
    ↳ TLink target, TLink newSource, TLink newTarget)
566 {
567     var equalityComparer = EqualityComparer<TLink>.Default;
568     var link = links.SearchOrDefault(source, target);
569     if (equalityComparer.Equals(link, default))
570     {
571         return links.CreateAndUpdate(newSource, newTarget);
572     }
573     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
    ↳ target))
574     {
575         return link;
576     }
577     return links.Update(link, newSource, newTarget);
578 }
579
580 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
581 /// <param name="links">Хранилище связей.</param>
582 /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>

```

```

583 /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
584 [MethodImpl(MethodImplOptions.AggressiveInlining)]
585 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
586 → target)
587 {
588     var link = links.SearchOrDefault(source, target);
589     if (!EqualityComparer<TLink>.Default.Equals(link, default))
590     {
591         links.Delete(link);
592         return link;
593     }
594     return default;
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 }

```

```

656 // TODO: Create a universal version of this method in Platform.Data (with using of for
657 ↪ loop)
658 public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
659 {
660     if (!links.AreValuesReset(linkIndex))
661     {
662         links.ResetValues(linkIndex);
663     }
664 }
665
666 /// <summary>
667 /// Merging two usages graphs, all children of old link moved to be children of new link
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)]

```

```

723     public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
724     ↪ TLink newLinkIndex)
725     {
726         var equalityComparer = EqualityComparer<TLink>.Default;
727         if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
728         {
729             links.MergeUsages(oldLinkIndex, newLinkIndex);
730             links.Delete(oldLinkIndex);
731         }
732         return newLinkIndex;
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 }

```

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

```

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

```

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

```

1  using System.Collections.Generic;
2  using Platform.Incrementers;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Incrementers
7  {
8      public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
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 }

```

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

```

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

```



```

6 namespace Platform.Data.Doublets.Incremeters
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         public TLink Increment(TLink unaryNumber)
19         {
20             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
21             {
22                 return Links.GetOrCreate(_unaryOne, _unaryOne);
23             }
24             var source = Links.GetSource(unaryNumber);
25             var target = Links.GetTarget(unaryNumber);
26             if (_equalityComparer.Equals(source, target))
27             {
28                 return Links.GetOrCreate(unaryNumber, _unaryOne);
29             }
30             else
31             {
32                 return Links.GetOrCreate(source, Increment(target));
33             }
34         }
35     }
36 }

```

1.22 ./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             {

```

```

45         SetValues(otherList, out Index, out Source, out Target);
46     }
47     else
48     {
49         throw new NotSupportedException();
50     }
51 }
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 public Link(ref Link<TLink> other) => SetValues(ref other, out Index, out Source, out
    ↳ Target);
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 public Link(TLink index, TLink source, TLink target)
58 {
59     Index = index;
60     Source = source;
61     Target = target;
62 }
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 private static void SetValues(ref Link<TLink> other, out TLink index, out TLink source,
    ↳ out TLink target)
66 {
67     index = other.Index;
68     source = other.Source;
69     target = other.Target;
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 private static void SetValues(IList<TLink> values, out TLink index, out TLink source,
    ↳ out TLink target)
74 {
75     switch (values.Count)
76     {
77         case 3:
78             index = values[0];
79             source = values[1];
80             target = values[2];
81             break;
82         case 2:
83             index = values[0];
84             source = values[1];
85             target = default;
86             break;
87         case 1:
88             index = values[0];
89             source = default;
90             target = default;
91             break;
92         default:
93             index = default;
94             source = default;
95             target = default;
96             break;
97     }
98 }
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 public override int GetHashCode() => (Index, Source, Target).GetHashCode();
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
105     && _equalityComparer.Equals(Source, _constants.Null)
106     && _equalityComparer.Equals(Target, _constants.Null);
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 public override bool Equals(object other) => other is Link<TLink> &&
    ↳ Equals((Link<TLink>)other);
110
111 [MethodImpl(MethodImplOptions.AggressiveInlining)]
112 public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
113     && _equalityComparer.Equals(Source, other.Source)
114     && _equalityComparer.Equals(Target, other.Target);
115
116 [MethodImpl(MethodImplOptions.AggressiveInlining)]
117 public static string ToString(TLink index, TLink source, TLink target) => $"({index}:
    ↳ {source}->{target})";
118
119 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

```

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

```

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

```

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

```

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

```

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

```

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

```

1 using System.Collections.Generic;
2 using Platform.Reflection;
3 using Platform.Converters;
4 using Platform.Numbers;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Numbers.Unary
9 {
10     public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
11         IConverter<TLink>
12     {
13     }
14 }

```

```

12     private static readonly EqualityComparer<TLink> _equalityComparer =
13         ↳ EqualityComparer<TLink>.Default;
14
15     private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
16
17     public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
18         ↳ powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
19         ↳ powerOf2ToUnaryNumberConverter;
20
21     public TLink Convert(TLink number)
22     {
23         var nullConstant = Links.Constants.Null;
24         var one = Integer<TLink>.One;
25         var target = nullConstant;
26         for (int i = 0; !_equalityComparer.Equals(number, default) && i <
27             ↳ NumericType<TLink>.BitsSize; i++)
28         {
29             if (_equalityComparer.Equals(Bit.And(number, one), one))
30             {
31                 target = _equalityComparer.Equals(target, nullConstant)
32                     ? _powerOf2ToUnaryNumberConverter.Convert(i)
33                     : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
34             }
35             number = Bit.ShiftRight(number, 1);
36         }
37         return target;
38     }
39 }

```

1.26 ./Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConveter.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 LinkToItsFrequencyNumberConveter<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 LinkToItsFrequencyNumberConveter(
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 }

```

1.27 ./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             _unaryNumberPowersOf2[0] = one;
22         }
23
24         public TLink Convert(int power)
25         {
26             Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
27                 ↪ - 1), nameof(power));
28             if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
29             {
30                 return _unaryNumberPowersOf2[power];
31             }
32             var previousPowerOf2 = Convert(power - 1);
33             var powerOf2 = Links.GetOrCreate(previousPowerOf2, previousPowerOf2);
34             _unaryNumberPowersOf2[power] = powerOf2;
35             return powerOf2;
36         }
37     }
38 }

```

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

```

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

```

```

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 *
74     ↪ 2UL);
75 }

```

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

```

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

```

```

39         break;
40     }
41     else
42     {
43         powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
44         SetBit(ref target, powerOf2Index);
45         source = Links.GetTarget(source);
46     }
47 }
48 }
49 return target;
50 }
51
52 [MethodImpl(MethodImplOptions.AggressiveInlining)]
53 private static void SetBit(ref TLink target, int powerOf2Index) => target =
    ↪ Bit.Or(target, Bit.ShiftLeft(Integer<TLink>.One, powerOf2Index));
54 }
55 }

```

1.30 ./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,
    ↪ TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↪ EqualityComparer<TLink>.Default;
12
13         public PropertiesOperator(ILinks<TLink> links) : base(links) { }
14
15         public TLink GetValue(TLink @object, TLink property)
16         {
17             var objectProperty = Links.SearchOrDefault(@object, property);
18             if (_equalityComparer.Equals(objectProperty, default))
19             {
20                 return default;
21             }
22             var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
23             if (valueLink == null)
24             {
25                 return default;
26             }
27             return Links.GetTarget(valueLink[Links.Constants.IndexPart]);
28         }
29
30         public void SetValue(TLink @object, TLink property, TLink value)
31         {
32             var objectProperty = Links.GetOrCreate(@object, property);
33             Links.DeleteMany(Links.AllIndices(Links.Constants.Any, objectProperty));
34             Links.GetOrCreate(objectProperty, value);
35         }
36     }
37 }

```

1.31 ./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 =
    ↪ EqualityComparer<TLink>.Default;
11
12         private readonly TLink _propertyMarker;
13         private readonly TLink _propertyValueMarker;
14
15         public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
    ↪ propertyValueMarker) : base(links)
16         {

```



```

17     _propertyMarker = propertyMarker;
18     _propertyValueMarker = propertyValueMarker;
19 }
20
21 public TLink Get(TLink link)
22 {
23     var property = Links.SearchOrDefault(link, _propertyMarker);
24     var container = GetContainer(property);
25     var value = GetValue(container);
26     return value;
27 }
28
29 private TLink GetContainer(TLink property)
30 {
31     var valueContainer = default(TLink);
32     if (_equalityComparer.Equals(property, default))
33     {
34         return valueContainer;
35     }
36     var constants = Links.Constants;
37     var countinueConstant = 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 countinueConstant;
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 }

```

1.32 ./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)

```

```

21 {
22     Links = links;
23     Header = header;
24     Break = constants.Break;
25     Continue = constants.Continue;
26 }
27
28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 protected abstract TLink GetTreeRoot();
30
31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 protected abstract TLink GetBasePartValue(TLink link);
33
34 [MethodImpl(MethodImplOptions.AggressiveInlining)]
35 protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
    ↪ rootSource, TLink rootTarget);
36
37 [MethodImpl(MethodImplOptions.AggressiveInlining)]
38 protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
    ↪ rootSource, TLink rootTarget);
39
40 [MethodImpl(MethodImplOptions.AggressiveInlining)]
41 protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
    ↪ AsRef<LinksHeader<TLink>>(Header);
42
43 [MethodImpl(MethodImplOptions.AggressiveInlining)]
44 protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
    ↪ AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes * (Integer<TLink>)link));
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
48 {
49     ref var link = ref GetLinkReference(linkIndex);
50     return new Link<TLink>(linkIndex, link.Source, link.Target);
51 }
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
55 {
56     ref var firstLink = ref GetLinkReference(first);
57     ref var secondLink = ref GetLinkReference(second);
58     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
59 }
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
63 {
64     ref var firstLink = ref GetLinkReference(first);
65     ref var secondLink = ref GetLinkReference(second);
66     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
67 }
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected virtual TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5,
    ↪ -5);
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 protected virtual void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
    ↪ Bit<TLink>.PartialWrite(storedValue, size, 5, -5);
74
75 [MethodImpl(MethodImplOptions.AggressiveInlining)]
76 protected virtual bool GetLeftIsChildValue(TLink value)
77 {
78     unchecked
79     {
80         //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 4, 1);
81         return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
82     }
83 }
84
85 [MethodImpl(MethodImplOptions.AggressiveInlining)]
86 protected virtual void SetLeftIsChildValue(ref TLink storedValue, bool value)
87 {
88     unchecked
89     {
90         var previousValue = storedValue;

```

```

91         var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 4,
92             ↪ 1);
93         storedValue = modified;
94     }
95 }
96 [MethodImpl(MethodImplOptions.AggressiveInlining)]
97 protected virtual bool GetRightIsChildValue(TLink value)
98 {
99     unchecked
100     {
101         //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 3, 1);
102         return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
103     }
104 }
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 protected virtual void SetRightIsChildValue(ref TLink storedValue, bool value)
108 {
109     unchecked
110     {
111         var previousValue = storedValue;
112         var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 3,
113             ↪ 1);
114         storedValue = modified;
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
132             ↪ end of sbyte
133         return (sbyte)value;
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
    ↳ (концом).
178 /// </summary>
179 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
180 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
181 /// <returns>Индекс искомой связи.</returns>
182 public TLink Search(TLink source, TLink target)
183 {
184     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)) //
            ↳ node.Key < root.Key
191         {
192             root = GetLeftOrDefault(root);
193         }
194         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
            ↳ node.Key > root.Key
195         {
196             root = GetRightOrDefault(root);
197         }
198         else // node.Key == root.Key
199         {
200             return root;
201         }
202     }
203     return Zero;
204 }
205
206 // TODO: Return indices range instead of references count
207 public TLink CountUsages(TLink link)
208 {
209     var root = GetTreeRoot();
210     var total = GetSize(root);
211     var totalRightIgnore = Zero;
212     while (!EqualToZero(root))
213     {
214         var @base = GetBasePartValue(root);
215         if (LessOrEqualThan(@base, link))
216         {
217             root = GetRightOrDefault(root);
218         }
219         else
220         {
221             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
222             root = GetLeftOrDefault(root);
223         }
224     }
225     root = GetTreeRoot();
226     var totalLeftIgnore = Zero;
227     while (!EqualToZero(root))
228     {
229         var @base = GetBasePartValue(root);
230         if (GreaterOrEqualThan(@base, link))
231         {
232             root = GetLeftOrDefault(root);
233         }
234         else
235         {
236             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
237             root = GetRightOrDefault(root);
238         }
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 }

```

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

```

```

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

```

```

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

```

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

```

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

```



```

16 [MethodImpl(MethodImplOptions.AggressiveInlining)]
17 protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
18
19 [MethodImpl(MethodImplOptions.AggressiveInlining)]
20 protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
21
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) =>
32     ↳ GetSizeValue(GetLinkReference(node).SizeAsSource);
33
34 [MethodImpl(MethodImplOptions.AggressiveInlining)]
35 protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
36     ↳ GetLinkReference(node).SizeAsSource, size);
37
38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 protected override bool GetLeftIsChild(TLink node) =>
40     ↳ GetLeftIsChildValue(GetLinkReference(node).SizeAsSource);
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected override void SetLeftIsChild(TLink node, bool value) =>
44     ↳ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 protected override bool GetRightIsChild(TLink node) =>
48     ↳ GetRightIsChildValue(GetLinkReference(node).SizeAsSource);
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 protected override void SetRightIsChild(TLink node, bool value) =>
52     ↳ SetRightIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected override sbyte GetBalance(TLink node) =>
56     ↳ GetBalanceValue(GetLinkReference(node).SizeAsSource);
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
60     ↳ GetLinkReference(node).SizeAsSource, value);
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsSource;
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
70     ↳ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
71     ↳ (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
75     ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
76     ↳ (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
77
78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 protected override void ClearNode(TLink node)
80 {
81     ref var link = ref GetLinkReference(node);
82     link.LeftAsSource = Zero;
83     link.RightAsSource = Zero;
84     link.SizeAsSource = Zero;
85 }
86 }
87 }

```

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

```

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

```

```

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         [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) => GetLinkReference(node).SizeAsSource;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override void SetSize(TLink node, TLink size) =>
40             ↳ GetLinkReference(node).SizeAsSource = size;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsSource;
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
50             ↳ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
51             ↳ (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
55             ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
56             ↳ (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         protected override void ClearNode(TLink node)
60         {
61             ref var link = ref GetLinkReference(node);
62             link.LeftAsSource = Zero;
63             link.RightAsSource = Zero;
64             link.SizeAsSource = Zero;
65         }
66     }
67 }

```

1.36 ./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)]

```

```

12     protected unsafe override ref TLink GetLeftReference(TLink node) => ref
13         ↳ GetLinkReference(node).LeftAsTarget;
14
15     [MethodImpl(MethodImplOptions.AggressiveInlining)]
16     protected unsafe override ref TLink GetRightReference(TLink node) => ref
17         ↳ GetLinkReference(node).RightAsTarget;
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     protected override void SetLeft(TLink node, TLink left) =>
27         ↳ GetLinkReference(node).LeftAsTarget = left;
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     protected override void SetRight(TLink node, TLink right) =>
31         ↳ GetLinkReference(node).RightAsTarget = right;
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override TLink GetSize(TLink node) =>
35         ↳ GetSizeValue(GetLinkReference(node).SizeAsTarget);
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
39         ↳ GetLinkReference(node).SizeAsTarget, size);
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override bool GetLeftIsChild(TLink node) =>
43         ↳ GetLeftIsChildValue(GetLinkReference(node).SizeAsTarget);
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override void SetLeftIsChild(TLink node, bool value) =>
47         ↳ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override bool GetRightIsChild(TLink node) =>
51         ↳ GetRightIsChildValue(GetLinkReference(node).SizeAsTarget);
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override void SetRightIsChild(TLink node, bool value) =>
55         ↳ SetRightIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override sbyte GetBalance(TLink node) =>
59         ↳ GetBalanceValue(GetLinkReference(node).SizeAsTarget);
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
63         ↳ GetLinkReference(node).SizeAsTarget, value);
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsTarget;
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
73         ↳ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
74         ↳ (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
75
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
78         ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
79         ↳ (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
80
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     protected override void ClearNode(TLink node)
83     {
84         ref var link = ref GetLinkReference(node);
85         link.LeftAsTarget = Zero;
86         link.RightAsTarget = Zero;
87         link.SizeAsTarget = Zero;
88     }
89 }

```

74 }

1.37 ./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 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)]
36         protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsTarget;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override void SetSize(TLink node, TLink size) =>
40             ↳ GetLinkReference(node).SizeAsTarget = size;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsTarget;
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
50             ↳ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
51             ↳ (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
55             ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
56             ↳ (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         protected override void ClearNode(TLink node)
60         {
61             ref var link = ref GetLinkReference(node);
62             link.LeftAsTarget = Zero;
63             link.RightAsTarget = Zero;
64             link.SizeAsTarget = Zero;
65         }
66     }
67 }
```

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

```
1 using System;
2 using System.Runtime.CompilerServices;
3 using Platform.Singletons;
4 using Platform.Numbers;
5 using Platform.Memory;
6 using static System.Runtime.CompilerServices.Unsafe;
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>
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public ResizableDirectMemoryLinks(string address, long memoryReservationStep) : this(new
33             ↳ FileMappedResizableDirectMemory(address, memoryReservationStep),
34             ↳ memoryReservationStep) { }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
38             ↳ DefaultLinksSizeStep) { }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
42             ↳ memoryReservationStep) : this(memory, memoryReservationStep,
43             ↳ Default<LinksConstants<TLink>>.Instance, true) { }
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
47             ↳ memoryReservationStep, LinksConstants<TLink> constants, bool useAvlBasedIndex) :
48             ↳ base(memory, memoryReservationStep, constants)
49         {
50             if (useAvlBasedIndex)
51             {
52                 _createSourceTreeMethods = () => new
53                     ↳ LinksSourcesAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
54                 _createTargetTreeMethods = () => new
55                     ↳ LinksTargetsAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
56             }
57             else
58             {
59                 _createSourceTreeMethods = () => new
60                     ↳ LinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
61                 _createTargetTreeMethods = () => new
62                     ↳ LinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
63             }
64             Init(memory, memoryReservationStep);
65         }
66
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         protected override void SetPointers(IResizableDirectMemory memory)
69         {
70             _links = (byte*)memory.Pointer;
71             _header = _links;
72             SourcesTreeMethods = _createSourceTreeMethods();
73             TargetsTreeMethods = _createTargetTreeMethods();
74             UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_links, _header);
75         }
76
77         [MethodImpl(MethodImplOptions.AggressiveInlining)]
78         protected override void ResetPointers()
79         {
80             base.ResetPointers();
81             _links = null;
82             _header = null;
83         }
84
85         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

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

```

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

```

```

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

```

```

140         {
141             return SourcesTreeMethods.CountUsages(source);
142         }
143         else //if(source != Any && target != Any)
144         {
145             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
146             var link = SourcesTreeMethods.Search(source, target);
147             return AreEqual(link, constants.Null) ? GetZero() : GetOne();
148         }
149     }
150     else
151     {
152         if (!Exists(index))
153         {
154             return GetZero();
155         }
156         if (AreEqual(source, any) && AreEqual(target, any))
157         {
158             return GetOne();
159         }
160         ref var storedLinkValue = ref GetLinkReference(index);
161         if (!AreEqual(source, any) && !AreEqual(target, any))
162         {
163             if (AreEqual(storedLinkValue.Source, source) &&
164                 ⇨ AreEqual(storedLinkValue.Target, target))
165             {
166                 return GetOne();
167             }
168             return GetZero();
169         }
170         var value = default(TLink);
171         if (AreEqual(source, any))
172         {
173             value = target;
174         }
175         if (AreEqual(target, any))
176         {
177             value = source;
178         }
179         if (AreEqual(storedLinkValue.Source, value) ||
180             ⇨ AreEqual(storedLinkValue.Target, value))
181         {
182             return GetOne();
183         }
184         return GetZero();
185     }
186 }
187
188 [MethodImpl(MethodImplOptions.AggressiveInlining)]
189 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
190 {
191     var constants = Constants;
192     var @break = constants.Break;
193     if (restrictions.Count == 0)
194     {
195         for (var link = GetOne(); LessOrEqualThan(link,
196             ⇨ GetHeaderReference().AllocatedLinks); link = Increment(link))
197         {
198             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
199             {
200                 return @break;
201             }
202         }
203         return @break;
204     }
205     var @continue = constants.Continue;
206     var any = constants.Any;
207     var index = restrictions[constants.IndexPart];
208     if (restrictions.Count == 1)
209     {
210         if (AreEqual(index, any))
211         {
212             return Each(handler, GetEmptyList());
213         }
214         if (!Exists(index))

```



```

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

```

```

291         AreEqual(storedLinkValue.Target, target))
292     {
293         return handler(GetLinkStruct(index));
294     }
295     return @continue;
296 }
297 var value = default(TLink);
298 if (AreEqual(source, any))
299 {
300     value = target;
301 }
302 if (AreEqual(target, any))
303 {
304     value = source;
305 }
306 if (AreEqual(storedLinkValue.Source, value) ||
307     AreEqual(storedLinkValue.Target, value))
308 {
309     return handler(GetLinkStruct(index));
310 }
311 return @continue;
312 }
313 }
314 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
315 }
316
317 /// <remarks>
318 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↳ в другом месте (но не в менеджере памяти, а в логике Links)
319 /// </remarks>
320 [MethodImpl(MethodImplOptions.AggressiveInlining)]
321 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
322 {
323     var constants = Constants;
324     var @null = constants.Null;
325     var linkIndex = restrictions[constants.IndexPart];
326     ref var link = ref GetLinkReference(linkIndex);
327     ref var header = ref GetHeaderReference();
328     ref var firstAsSource = ref header.FirstAsSource;
329     ref var firstAsTarget = ref header.FirstAsTarget;
330     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↳ предварительно заполнено нулями
331     if (!AreEqual(link.Source, @null))
332     {
333         SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
334     }
335     if (!AreEqual(link.Target, @null))
336     {
337         TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
338     }
339     link.Source = substitution[constants.SourcePart];
340     link.Target = substitution[constants.TargetPart];
341     if (!AreEqual(link.Source, @null))
342     {
343         SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
344     }
345     if (!AreEqual(link.Target, @null))
346     {
347         TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
348     }
349     return linkIndex;
350 }
351
352 /// <remarks>
353 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
    ↳ пространство
354 /// </remarks>
355 public virtual TLink Create(IList<TLink> restrictions)
356 {
357     ref var header = ref GetHeaderReference();
358     var freeLink = header.FirstFreeLink;
359     if (!AreEqual(freeLink, Constants.Null))
360     {
361         UnusedLinksListMethods.Detach(freeLink);
362     }
363     else
364     {
365         var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;

```

```

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

```

```

441         && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
442         && !IsUnusedLink(link);
443
444     [MethodImpl(MethodImplOptions.AggressiveInlining)]
445     protected virtual bool IsUnusedLink(TLink linkIndex)
446     {
447         if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
448             ↪ is not needed
449         {
450             ref var link = ref GetLinkReference(linkIndex);
451             return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
452         }
453         else
454         {
455             return true;
456         }
457     }
458
459     [MethodImpl(MethodImplOptions.AggressiveInlining)]
460     protected virtual TLink GetOne() => Integer<TLink>.One;
461
462     [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 GreaterThan(TLink first, TLink second) => Comparer.Compare(first,
479         ↪ second) > 0;
480
481     [MethodImpl(MethodImplOptions.AggressiveInlining)]
482     protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
483         ↪ Comparer.Compare(first, second) >= 0;
484
485     [MethodImpl(MethodImplOptions.AggressiveInlining)]
486     protected virtual long ConvertToUInt64(TLink value) => (Integer<TLink>)value;
487
488     [MethodImpl(MethodImplOptions.AggressiveInlining)]
489     protected virtual TLink ConvertToAddress(long value) => (Integer<TLink>)value;
490
491     [MethodImpl(MethodImplOptions.AggressiveInlining)]
492     protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
493         ↪ second);
494
495     [MethodImpl(MethodImplOptions.AggressiveInlining)]
496     protected virtual TLink Subtract(TLink first, TLink second) =>
497         ↪ Arithmetic<TLink>.Subtract(first, second);
498
499     [MethodImpl(MethodImplOptions.AggressiveInlining)]
500     protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
501
502     [MethodImpl(MethodImplOptions.AggressiveInlining)]
503     protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
504
505     [MethodImpl(MethodImplOptions.AggressiveInlining)]
506     protected virtual IList<TLink> GetEmptyList() => Array.Empty<TLink>();
507
508     #region Disposable
509
510     protected override bool AllowMultipleDisposeCalls => true;
511
512     protected override void Dispose(bool manual, bool wasDisposed)
513     {
514         if (!wasDisposed)
515         {
516             ResetPointers();
517             _memory.DisposeIfPossible();
518         }
519     }

```

```

512         #endregion
513     }
514 }
515 }

```

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

```

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

```

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

```

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

```

```

7         void Detach(TLink freeLink);
8         void AttachAsFirst(TLink link);
9     }
10 }

```

1.42 ./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 }

```

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

```

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

```

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

```

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

```

```

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

```

1.45 ./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
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override ulong GetZero() => OUL;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override bool EqualToZero(ulong value) => value == OUL;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override bool AreEqual(ulong first, ulong second) => first == second;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override bool GreaterThanZero(ulong value) => value > OUL;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override bool GreaterThan(ulong first, ulong second) => first > second;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;

```

```

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);
111
112 [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 }

```

1.46 ./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
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetZero() => 0UL;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override bool EqualToZero(ulong value) => value == 0UL;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override bool AreEqual(ulong first, ulong second) => first == second;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override bool GreaterThanZero(ulong value) => value > 0UL;
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected override bool GreaterThan(ulong first, ulong second) => first > second;
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
42         ↳ always true for ulong
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
46         ↳ always >= 0 for ulong
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
53         ↳ for ulong
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override bool LessThan(ulong first, ulong second) => first < second;
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         protected override ulong Increment(ulong value) => ++value;
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         protected override ulong Decrement(ulong value) => --value;
63     }
64 }

```

```

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

```

1.47 ./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)]
44         protected override bool GetLeftIsChild(ulong node) =>
45             ↪ GetLeftIsChildValue(Links[node].SizeAsSource);
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override bool GetRightIsChild(ulong node) =>
49             ↪ GetRightIsChildValue(Links[node].SizeAsSource);
50     }
51 }

```

```

37
38 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 // protected override bool GetLeftIsChild(ulong node) => IsChild(node, GetLeft(node));
40
41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 protected override void SetLeftIsChild(ulong node, bool value) =>
43     ↳ SetLeftIsChildValue(ref Links[node].SizeAsSource, value);
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 protected override bool GetRightIsChild(ulong node) =>
47     ↳ GetRightIsChildValue(Links[node].SizeAsSource);
48
49 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
50 // protected override bool GetRightIsChild(ulong node) => IsChild(node, GetRight(node));
51
52 [MethodImpl(MethodImplOptions.AggressiveInlining)]
53 protected override void SetRightIsChild(ulong node, bool value) =>
54     ↳ SetRightIsChildValue(ref Links[node].SizeAsSource, value);
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 protected override sbyte GetBalance(ulong node) =>
58     ↳ GetBalanceValue(Links[node].SizeAsSource);
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
62     ↳ Links[node].SizeAsSource, value);
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 protected override ulong GetTreeRoot() => Header->FirstAsSource;
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
72     ↳ ulong secondSource, ulong secondTarget)
73     => firstSource < secondSource || (firstSource == secondSource && firstTarget <
74     ↳ secondTarget);
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
78     ↳ ulong secondSource, ulong secondTarget)
79     => firstSource > secondSource || (firstSource == secondSource && firstTarget >
80     ↳ secondTarget);
81
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 protected override void ClearNode(ulong node)
84 {
85     ref var link = ref Links[node];
86     link.LeftAsSource = OUL;
87     link.RightAsSource = OUL;
88     link.SizeAsSource = OUL;
89 }
90 }
91 }
92 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
6 {
7     public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
8         ↳ UInt64LinksSizeBalancedTreeMethodsBase
9     {
10         public UInt64LinksSourcesSizeBalancedTreeMethods(LinksConstants<ulong> constants,
11             ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
12             ↳ { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetLeftReference(ulong node) => ref
16             ↳ Links[node].LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20             ↳ Links[node].RightAsSource;
21
22     }
23 }

```

```

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

```

1.49 ./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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
30             ↳ left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
34             ↳ right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
41             ↳ size;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override ulong GetTreeRoot() => Header->FirstAsTarget;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool FirstIsToTheLeftOfSecond(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 bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
57             ↳ ulong secondSource, ulong secondTarget)
58             ↳ => firstSource > secondSource || (firstSource == secondSource && firstTarget >
59             ↳ secondTarget);
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         protected override void ClearNode(ulong node)
63         {
64             ref var link = ref Links[node];
65             link.LeftAsTarget = OUL;
66             link.RightAsTarget = OUL;
67             link.SizeAsTarget = OUL;
68         }
69     }
70 }

```

```

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

```

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

```

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

```

```

9      public UInt64LinksTargetsSizeBalancedTreeMethods(LinksConstants<ulong> constants,
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 }

```

1.51 ./Platform.Data.Doublents/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.Doublents.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.Doublents.ResizableDirectMemory.Specific
11 {
12     public unsafe class UInt64ResizableDirectMemoryLinks : ResizableDirectMemoryLinksBase<ulong>
13     {
14         private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
15         private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
16         private LinksHeader<ulong>* _header;

```

```

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

```

```

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 }

```

1.52 ./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 }

```

1.53 ./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 private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
41 {
42     var loopedLength = length - (length % 2);
43     for (var i = 0; i < loopedLength; i += 2)
44     {
45         destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
46     }
47     if (length > loopedLength)
48     {
49         destination[length / 2] = source[length - 1];
50     }
51 }
52 }
53 }

```

1.54 ./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

```

```

33 {
34     public TLink Element;
35     public LinkFrequency<TLink> DoubletData;
36
37     public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
38     {
39         Element = element;
40         DoubletData = doubletData;
41     }
42
43     public override string ToString() => $"{Element}: ({DoubletData})";
44 }
45
46 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
47     : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One, true)
48 {
49 }
50
51 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
    ↪ doInitialFrequenciesIncrement)
52     : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One,
    ↪ doInitialFrequenciesIncrement)
53 {
54 }
55
56 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
    ↪ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
57     : base(links)
58 {
59     _baseConverter = baseConverter;
60     _doubletFrequenciesCache = doubletFrequenciesCache;
61     if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
62     {
63         minFrequencyToCompress = Integer<TLink>.One;
64     }
65     _minFrequencyToCompress = minFrequencyToCompress;
66     _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
67     ResetMaxDoublet();
68 }
69
70 public override TLink Convert(IList<TLink> source) =>
    ↪ _baseConverter.Convert(Compress(source));
71
72 /// <remarks>
73 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
74 /// Faster version (doublets' frequencies dictionary is not recreated).
75 /// </remarks>
76 private IList<TLink> Compress(IList<TLink> sequence)
77 {
78     if (sequence.IsNullOrEmpty())
79     {
80         return null;
81     }
82     if (sequence.Count == 1)
83     {
84         return sequence;
85     }
86     if (sequence.Count == 2)
87     {
88         return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
89     }
90     // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
91     var copy = new HalfDoublet[sequence.Count];
92     Doublet<TLink> doublet = default;
93     for (var i = 1; i < sequence.Count; i++)
94     {
95         doublet.Source = sequence[i - 1];
96         doublet.Target = sequence[i];
97         LinkFrequency<TLink> data;
98         if (_doInitialFrequenciesIncrement)
99         {
100             data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
101         }
102         else
103         {
104             data = _doubletFrequenciesCache.GetFrequency(ref doublet);

```

```

105         if (data == null)
106         {
107             throw new NotSupportedException("If you ask not to increment
                ↳ frequencies, it is expected that all frequencies for the sequence
                ↳ are prepared.");
108         }
109     }
110     copy[i - 1].Element = sequence[i - 1];
111     copy[i - 1].DoubletData = data;
112     UpdateMaxDoublet(ref doublet, data);
113 }
114 copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
115 copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
116 if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
117 {
118     var newLength = ReplaceDoublets(copy);
119     sequence = new TLink[newLength];
120     for (int i = 0; i < newLength; i++)
121     {
122         sequence[i] = copy[i].Element;
123     }
124 }
125 return sequence;
126 }
127
128 /// <remarks>
129 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
130 /// </remarks>
131 private int ReplaceDoublets(HalfDoublet[] copy)
132 {
133     var oldLength = copy.Length;
134     var newLength = copy.Length;
135     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
136     {
137         var maxDoubletSource = _maxDoublet.Source;
138         var maxDoubletTarget = _maxDoublet.Target;
139         if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
140         {
141             _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
142         }
143         var maxDoubletReplacementLink = _maxDoubletData.Link;
144         oldLength--;
145         var oldLengthMinusTwo = oldLength - 1;
146         // Substitute all usages
147         int w = 0, r = 0; // (r == read, w == write)
148         for (; r < oldLength; r++)
149         {
150             if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
                ↳ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
151             {
152                 if (r > 0)
153                 {
154                     var previous = copy[w - 1].Element;
155                     copy[w - 1].DoubletData.DecrementFrequency();
156                     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,
                        ↳ 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 runderom
218     ↪ numbers data (but gives collisions anyway) */
219     {
220         _maxDoublet = doublet;
221         _maxDoubletData = data;
222     }
223 }
224 }
225 }
226 }
227 }

```

1.55 ./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>,
9     ↪ TLink>
10     {
11         protected readonly IList<TLink> Links;
12         public LinksListToSequenceConverterBase(IList<TLink> links) => Links = links;
13         public abstract TLink Convert(IList<TLink> source);
14     }
15 }

```

1.56 ./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 =
12     ↳ EqualityComparer<TLink>.Default;
13 private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
14 private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
15
16 public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
17     ↳ sequenceToItsLocalElementLevelsConverter) : base(links)
18     => _sequenceToItsLocalElementLevelsConverter =
19         ↳ sequenceToItsLocalElementLevelsConverter;
20
21 public override TLink Convert(IList<TLink> sequence)
22 {
23     var length = sequence.Count;
24     if (length == 1)
25     {
26         return sequence[0];
27     }
28     var links = Links;
29     if (length == 2)
30     {
31         return links.GetOrCreate(sequence[0], sequence[1]);
32     }
33     sequence = sequence.ToArray();
34     var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
35     while (length > 2)
36     {
37         var levelRepeat = 1;
38         var currentLevel = levels[0];
39         var previousLevel = levels[0];
40         var skipOnce = false;
41         var w = 0;
42         for (var i = 1; i < length; i++)
43         {
44             if (_equalityComparer.Equals(currentLevel, levels[i]))
45             {
46                 levelRepeat++;
47                 skipOnce = false;
48                 if (levelRepeat == 2)
49                 {
50                     sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
51                     var newLevel = i >= length - 1 ?
52                         GetPreviousLowerThanCurrentOrCurrent(previousLevel,
53                             ↳ currentLevel) :
54                         i < 2 ?
55                             GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
56                             GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
57                                 ↳ currentLevel, levels[i + 1]);
58                     levels[w] = newLevel;
59                     previousLevel = currentLevel;
60                     w++;
61                     levelRepeat = 0;
62                     skipOnce = true;
63                 }
64             }
65             else if (i == length - 1)
66             {
67                 sequence[w] = sequence[i];
68                 levels[w] = levels[i];
69                 w++;
70             }
71         }
72     }
73     else
74     {
75         currentLevel = levels[i];
76         levelRepeat = 1;
77         if (skipOnce)
78         {
79             skipOnce = false;
80         }
81         else
82         {
83             sequence[w] = sequence[i - 1];
84             levels[w] = levels[i - 1];
85             previousLevel = levels[w];
86             w++;
87         }
88     }
89     if (i == length - 1)
90     {
91         sequence[w] = sequence[i];
92     }
93 }

```

```

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 }

```

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

```

1.58 ./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 }

```

1.59 ./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 =
11             ↳ EqualityComparer<TLink>.Default;
12
13         private readonly ILinks<TLink> _links;
14         private readonly TLink _sequenceMarkerLink;
15
16         public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
17         {
18             _links = links;
19             _sequenceMarkerLink = sequenceMarkerLink;
20
21             public bool IsMatched(TLink sequenceCandidate)
22                 => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
23                 || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
24                     ↳ sequenceCandidate), _links.Constants.Null);
25     }
26 }
```

1.60 ./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>,
11         ↳ ISequenceAppender<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↳ EqualityComparer<TLink>.Default;
15
16         private readonly IStack<TLink> _stack;
17         private readonly ISequenceHeightProvider<TLink> _heightProvider;
18
19         public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
20             ↳ ISequenceHeightProvider<TLink> heightProvider)
21             : base(links)
22         {
23             _stack = stack;
24             _heightProvider = heightProvider;
25
26             public TLink Append(TLink sequence, TLink appendant)
27             {
28                 var cursor = sequence;
29                 while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
30                 {
31                     var source = Links.GetSource(cursor);
32                     var target = Links.GetTarget(cursor);
33                     if (_equalityComparer.Equals(_heightProvider.Get(source),
34                         ↳ _heightProvider.Get(target)))
35                     {
36                         break;
37                     }
38                     else
39                     {
40                         _stack.Push(source);
41                         cursor = target;
42                     }
43                 }
44                 var left = cursor;
45                 var right = appendant;
46                 while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
47                 {
48                     right = Links.GetOrCreate(left, right);
49                     left = cursor;
50                 }
51             }
52         }
53     }
```

```

47         }
48         return Links.GetOrCreate(left, right);
49     }
50 }
51 }

```

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

```

1 using System.Collections.Generic;
2 using System.Linq;
3 using Platform.Interfaces;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Sequences
8 {
9     public class DuplicateSegmentsCounter<TLink> : ICounter<int>
10     {
11         private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
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 }

```

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

```

1 using System;
2 using System.Linq;
3 using System.Collections.Generic;
4 using Platform.Interfaces;
5 using Platform.Collections;
6 using Platform.Collections.Lists;
7 using Platform.Collections.Segments;
8 using Platform.Collections.Segments.Walkers;
9 using Platform.Singletons;
10 using Platform.Numbers;
11 using Platform.Data.Doublets.Unicode;
12
13 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14
15 namespace Platform.Data.Doublets.Sequences
16 {
17     public class DuplicateSegmentsProvider<TLink> :
18         ↪ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
19         ↪ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
20     {
21         private readonly ILinks<TLink> _links;
22         private readonly ILinks<TLink> _sequences;
23         private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
24         private BitString _visited;
25
26         private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
27             ↪ IList<TLink>>>
28         {
29             private readonly IListEqualityComparer<TLink> _listComparer;
30             public ItemEquilityComparer() => _listComparer =
31                 ↪ Default<IListEqualityComparer<TLink>>.Instance;
32             public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
33                 ↪ KeyValuePair<IList<TLink>, IList<TLink>> right) =>
34                 ↪ _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
35                 ↪ right.Value);
36             public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
37                 ↪ (_listComparer.GetHashCode(pair.Key),
38                 ↪ _listComparer.GetHashCode(pair.Value)).GetHashCode();
39         }
40
41         private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
42         {
43             private readonly IListComparer<TLink> _listComparer;
44
45             public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
46
47             public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
48                 ↪ KeyValuePair<IList<TLink>, IList<TLink>> right)
49             {
50                 var intermediateResult = _listComparer.Compare(left.Key, right.Key);
51                 if (intermediateResult == 0)
52                 {
53                     intermediateResult = _listComparer.Compare(left.Value, right.Value);
54                 }
55             }
56         }
57     }
58 }

```



```

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>,
59         ↪ IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
60     var count = _links.Count();
61     _visited = new BitString((long)(Integer<TLink>)count + 1);
62     _links.Each(link =>
63     {
64         var linkIndex = _links.GetIndex(link);
65         var linkBitIndex = (long)(Integer<TLink>)linkIndex;
66         if (!_visited.Get(linkBitIndex))
67         {
68             var sequenceElements = new List<TLink>();
69             var filler = new ListFiller<TLink, TLink>(sequenceElements,
70                 ↪ _sequences.Constants.Break);
71             _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
72                 ↪ LinkAddress<TLink>(linkIndex));
73             if (sequenceElements.Count > 2)
74             {
75                 WalkAll(sequenceElements);
76             }
77         }
78         return _links.Constants.Continue;
79     });
80     var resultList = _groups.ToList();
81     var comparer = Default<ItemComparer>.Instance;
82     resultList.Sort(comparer);
83
84 #if DEBUG
85     foreach (var item in resultList)
86     {
87         PrintDuplicates(item);
88     }
89 #endif
90     return resultList;
91 }
92
93 protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
94     ↪ length) => new Segment<TLink>(elements, offset, length);
95
96 protected override void OnDuplicateFound(Segment<TLink> segment)
97 {
98     var duplicates = CollectDuplicatesForSegment(segment);
99     if (duplicates.Count > 1)
100     {
101         _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
102             ↪ duplicates));
103     }
104 }
105
106 private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
107 {
108     var duplicates = new List<TLink>();
109     var readAsElement = new HashSet<TLink>();
110     var restrictions = segment.ShiftRight();
111     restrictions[0] = _sequences.Constants.Any;
112     _sequences.Each(sequence =>
113     {
114         var sequenceIndex = sequence[_sequences.Constants.IndexPart];
115         duplicates.Add(sequenceIndex);
116         readAsElement.Add(sequenceIndex);
117         return _sequences.Constants.Continue;
118     }, restrictions);
119     if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
120     {
121         return new List<TLink>();
122     }
123 }

```

```

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
137     private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
138     {
139         if (!(_links is ILinks<ulong> ulongLinks))
140         {
141             return;
142         }
143         var duplicatesKey = duplicatesItem.Key;
144         var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
145         Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
146         var duplicatesList = duplicatesItem.Value;
147         for (int i = 0; i < duplicatesList.Count; i++)
148         {
149             ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
150             var formattedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
151                 ↪ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
152                 ↪ UnicodeMap.IsCharLink(link.Index) ?
153                 ↪ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
154             Console.WriteLine(formattedSequenceStructure);
155             var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
156                 ↪ ulongLinks);
157             Console.WriteLine(sequenceString);
158         }
159         Console.WriteLine();
160     }
161 }

```

1.63 ./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     }
33 }

```

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

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

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

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

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

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

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

public void ValidateFrequencies()
{
    foreach (var entry in _doubletsCache)
    {
        var value = entry.Value;
        var linkIndex = value.Link;
        if (!_equalityComparer.Equals(linkIndex, default))
        {
            var frequency = value.Frequency;
            var count = _frequencyCounter.Count(linkIndex);
            // TODO: Why `frequency` always greater than `count` by 1?
            if (((_comparer.Compare(frequency, count) > 0) &&
                ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count),
                ↪ Integer<TLink>.One) > 0))
```

```

105         || (_comparer.Compare(count, frequency) > 0) &&
           ↳ (_comparer.Compare(Arithmetic.Subtract(count, frequency),
           ↳ Integer<TLink>.One) > 0)))
106     {
107         throw new InvalidOperationException("Frequencies validation failed.");
108     }
109 }
110 //else
111 //{
112 //    if (value.Frequency > 0)
113 //    {
114 //        var frequency = value.Frequency;
115 //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
116 //        var count = _countLinkFrequency(linkIndex);
117 //
118 //        if ((frequency > count && frequency - count > 1) || (count > frequency
           ↳ && count - frequency > 1))
119 //            throw new Exception("Frequencies validation failed.");
120 //    }
121 //}
122 }
123 }
124 }
125 }

```

1.64 ./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 }

```

1.65 ./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> :
           ↳ IConverter<Doublet<TLink>, TLink>
8     {
9         private readonly LinkFrequenciesCache<TLink> _cache;
10         public
           ↳ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
           ↳ cache) => _cache = cache;
11         public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
12     }
13 }

```

1.66 ./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 }

```

1.67 ./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                 return _total;
35             }
36             StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
37                 ↳ IsElement, VisitElement);
38             return _total;
39         }
40
41         private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
42             ↳ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
43             ↳ IsPartialPoint
44
45         private bool VisitElement(TLink element)
46         {
47             if (_equalityComparer.Equals(element, _symbol))
48             {
49                 _total = Arithmetic.Increment(_total);
50             }
51             return true;
52         }
53     }
54 }

```

1.68 ./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 }
```

1.69 ./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 }
```

1.70 ./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    }
14 }
```

1.71 ./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 }

```

1.72 ./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;

```

```

18
19     public CachedSequenceHeightProvider(
20         ILinks<TLink> links,
21         ISequenceHeightProvider<TLink> baseHeightProvider,
22         IConverter<TLink> addressToUnaryNumberConverter,
23         IConverter<TLink> unaryNumberToAddressConverter,
24         TLink heightPropertyMarker,
25         IProperties<TLink, TLink, TLink> propertyOperator)
26         : base(links)
27     {
28         _heightPropertyMarker = heightPropertyMarker;
29         _baseHeightProvider = baseHeightProvider;
30         _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
31         _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
32         _propertyOperator = propertyOperator;
33     }
34
35     public TLink Get(TLink sequence)
36     {
37         TLink height;
38         var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
39         if (_equalityComparer.Equals(heightValue, default))
40         {
41             height = _baseHeightProvider.Get(sequence);
42             heightValue = _addressToUnaryNumberConverter.Convert(height);
43             _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
44         }
45         else
46         {
47             height = _unaryNumberToAddressConverter.Convert(heightValue);
48         }
49         return height;
50     }
51 }
52

```

1.73 ./Platform.Data.Doublents/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.Doublents.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             {
22                 pairOrElement = Links.GetTarget(pairOrElement);
23                 height = Arithmetic.Increment(height);
24             }
25             return height;
26         }
27     }
28 }
29

```

1.74 ./Platform.Data.Doublents/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.Doublents.Sequences.HeightProviders
6 {
7     public interface ISequenceHeightProvider<TLink> : IProvider<TLink, TLink>
8     {
9     }
10 }

```


1.75 ./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             for (; i >= 1; i--)
25             {
26                 _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
27             }
28             return indexed;
29         }
30
31         private bool IsIndexedWithIncrement(TLink source, TLink target)
32         {
33             var frequency = _cache.GetFrequency(source, target);
34             if (frequency == null)
35             {
36                 return false;
37             }
38             var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
39             if (indexed)
40             {
41                 _cache.IncrementFrequency(source, target);
42             }
43             return indexed;
44         }
45
46         public bool MightContain(IList<TLink> sequence)
47         {
48             var indexed = true;
49             var i = sequence.Count;
50             while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
51             return indexed;
52         }
53
54         private bool IsIndexed(TLink source, TLink target)
55         {
56             var frequency = _cache.GetFrequency(source, target);
57             if (frequency == null)
58             {
59                 return false;
60             }
61             return !_equalityComparer.Equals(frequency.Frequency, default);
62         }
63     }
64 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Increments;
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 }

```

```

12
13     private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
14     private readonly IIncrementer<TLink> _frequencyIncrementer;
15
16     public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IProperty<TLink, TLink>
17     ↪ frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
18         : base(links)
19     {
20         _frequencyPropertyOperator = frequencyPropertyOperator;
21         _frequencyIncrementer = frequencyIncrementer;
22     }
23
24     public override bool Add(IList<TLink> sequence)
25     {
26         var indexed = true;
27         var i = sequence.Count;
28         while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
29             ↪ { }
30         for (; i >= 1; i--)
31         {
32             Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
33         }
34         return indexed;
35     }
36
37     private bool IsIndexedWithIncrement(TLink source, TLink target)
38     {
39         var link = Links.SearchOrDefault(source, target);
40         var indexed = !_equalityComparer.Equals(link, default);
41         if (indexed)
42         {
43             Increment(link);
44         }
45         return indexed;
46     }
47
48     private void Increment(TLink link)
49     {
50         var previousFrequency = _frequencyPropertyOperator.Get(link);
51         var frequency = _frequencyIncrementer.Increment(previousFrequency);
52         _frequencyPropertyOperator.Set(link, frequency);
53     }
54 }

```

1.77 ./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 }

```

1.78 ./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 }

```

```

13     public virtual bool Add(IList<TLink> sequence)
14     {
15         var indexed = true;
16         var i = sequence.Count;
17         while (--i >= 1 && (indexed =
            ↳ !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
            ↳ default))) { }
18         for (; i >= 1; i--)
19         {
20             Links.GetOrCreate(sequence[i - 1], sequence[i]);
21         }
22         return indexed;
23     }
24
25     public virtual bool MightContain(IList<TLink> sequence)
26     {
27         var indexed = true;
28         var i = sequence.Count;
29         while (--i >= 1 && (indexed =
            ↳ !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
            ↳ default))) { }
30         return indexed;
31     }
32 }
33 }

```

1.79 ./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 =
            ↳ EqualityComparer<TLink>.Default;
10
11         private readonly ISynchronizedLinks<TLink> _links;
12
13         public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;
14
15         public bool Add(IList<TLink> sequence)
16         {
17             var indexed = true;
18             var i = sequence.Count;
19             var links = _links.Unsync;
20             _links.SyncRoot.ExecuteReadOperation(() =>
21             {
22                 while (--i >= 1 && (indexed =
                    ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                    ↳ sequence[i]), default))) { }
23             });
24             if (!indexed)
25             {
26                 _links.SyncRoot.ExecuteWriteOperation(() =>
27                 {
28                     for (; i >= 1; i--)
29                     {
30                         links.GetOrCreate(sequence[i - 1], sequence[i]);
31                     }
32                 });
33             }
34             return indexed;
35         }
36
37         public bool MightContain(IList<TLink> sequence)
38         {
39             var links = _links.Unsync;
40             return _links.SyncRoot.ExecuteReadOperation(() =>
41             {
42                 var indexed = true;
43                 var i = sequence.Count;
44                 while (--i >= 1 && (indexed =
                    ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                    ↳ sequence[i]), default))) { }
45                 return indexed;
46             });
47         }
48     }
49 }

```

```

48     }
49 }

```

1.80 ./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 }

```

1.81 ./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 Array.Empty<ulong>();
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.GetOrCreate(sequence[startAt], sequence[stopAt]) };
60                 }
61                 var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];

```

```

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

```

```

135 #endregion
136
137 public HashSet<ulong> Each1(params ulong[] sequence)
138 {
139     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
140     Each1(link =>
141     {
142         if (!visitedLinks.Contains(link))
143         {
144             visitedLinks.Add(link); // изучить почему случаются повторы
145         }
146         return true;
147     }, sequence);
148     return visitedLinks;
149 }
150
151 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
152 {
153     if (sequence.Length == 2)
154     {
155         Links.Unsync.Each(sequence[0], sequence[1], handler);
156     }
157     else
158     {
159         var innerSequenceLength = sequence.Length - 1;
160         for (var li = 0; li < innerSequenceLength; li++)
161         {
162             var left = sequence[li];
163             var right = sequence[li + 1];
164             if (left == 0 && right == 0)
165             {
166                 continue;
167             }
168             var linkIndex = li;
169             ulong[] innerSequence = null;
170             Links.Unsync.Each(doublet =>
171             {
172                 if (innerSequence == null)
173                 {
174                     innerSequence = new ulong[innerSequenceLength];
175                     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[]
↪ sequence)
223 {
224     if (sequence.IsNullOrEmpty())
225     {
226         return;
227     }
228     Links.EnsureLinkIsAnyOrExists(sequence);
229     if (sequence.Length == 1)
230     {
231         var link = sequence[0];
232         if (link > 0)
233         {
234             handler(new LinkAddress<LinkIndex>(link));
235         }
236         else
237         {
238             Links.Each(Constants.Any, Constants.Any, handler);
239         }
240     }
241     else if (sequence.Length == 2)
242     {
243         // _links.Each(sequence[0], sequence[1], handler);
244         //   o_|      x_o ...
245         // x_|      |__|
246         Links.Each(sequence[1], Constants.Any, doublet =>
247         {
248             var match = Links.SearchOrDefault(sequence[0], doublet);
249             if (match != Constants.Null)
250             {
251                 handler(new LinkAddress<LinkIndex>(match));
252             }
253             return true;
254         });
255         // |_x      ... x_o
256         // |_o      |__|
257         Links.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
    ↪ stepFrom)
299 {
300     var upStep = stepFrom;
301     var firstSource = Links.Unsync.GetTarget(upStep);
302     while (firstSource != right && firstSource != upStep)
303     {
304         upStep = firstSource;
305         firstSource = Links.Unsync.GetSource(upStep);
306     }
307     if (firstSource == right)
308     {
309         handler(new LinkAddress<LinkIndex>(stepFrom));
310     }
311 }
312
313 // TODO: Test
314 private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
315 {
316     Links.Unsync.Each(right, Constants.Any, doublet =>
317     {
318         StepLeft(handler, left, doublet);
319         if (right != doublet)
320         {
321             PartialStepLeft(handler, left, doublet);
322         }
323         return true;
324     });
325 }
326
327 private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
328 {
329     Links.Unsync.Each(Constants.Any, right, leftStep =>
330     {
331         TryStepLeftUp(handler, left, leftStep);
332         return true;
333     });
334 }
335
336 private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
337 {
338     var upStep = stepFrom;
339     var firstTarget = Links.Unsync.GetSource(upStep);
340     while (firstTarget != left && firstTarget != upStep)
341     {
342         upStep = firstTarget;
343         firstTarget = Links.Unsync.GetTarget(upStep);
344     }
345     if (firstTarget == left)
346     {
347         handler(new LinkAddress<LinkIndex>(stepFrom));
348     }
349 }
350
351 private bool StartsWith(ulong sequence, ulong link)
352 {
353     var upStep = sequence;
354     var firstSource = Links.Unsync.GetSource(upStep);
355     while (firstSource != link && firstSource != upStep)
356     {
357         upStep = firstSource;
358         firstSource = Links.Unsync.GetSource(upStep);
359     }
360     return firstSource == link;
361 }
362
363 private bool EndsWith(ulong sequence, ulong link)
364 {
365     var upStep = sequence;
366     var lastTarget = Links.Unsync.GetTarget(upStep);
367     while (lastTarget != link && lastTarget != upStep)
368     {

```



```

369         upStep = lastTarget;
370         lastTarget = Links.Unsync.GetTarget(upStep);
371     }
372     return lastTarget == link;
373 }
374
375 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
376 {
377     return _sync.ExecuteReadOperation(() =>
378     {
379         var results = new List<ulong>();
380         if (sequence.Length > 0)
381         {
382             Links.EnsureLinkExists(sequence);
383             var firstElement = sequence[0];
384             if (sequence.Length == 1)
385             {
386                 results.Add(firstElement);
387                 return results;
388             }
389             if (sequence.Length == 2)
390             {
391                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
392                 if (doublet != Constants.Null)
393                 {
394                     results.Add(doublet);
395                 }
396                 return results;
397             }
398             var linksInSequence = new HashSet<ulong>(sequence);
399             void handler(IList<LinkIndex> result)
400             {
401                 var resultIndex = result[Links.Constants.IndexPart];
402                 var filterPosition = 0;
403                 StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
404                     ↪ Links.Unsync.GetTarget,
405                     x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
406                     ↪ x =>
407                     {
408                         if (filterPosition == sequence.Length)
409                         {
410                             filterPosition = -2; // Длиннее чем нужно
411                             return false;
412                         }
413                         if (x != sequence[filterPosition])
414                         {
415                             filterPosition = -1;
416                             return false; // Начинается иначе
417                         }
418                         filterPosition++;
419                         return true;
420                     });
421                 if (filterPosition == sequence.Length)
422                 {
423                     results.Add(resultIndex);
424                 }
425             }
426             if (sequence.Length >= 2)
427             {
428                 StepRight(handler, sequence[0], sequence[1]);
429             }
430             var last = sequence.Length - 2;
431             for (var i = 1; i < last; i++)
432             {
433                 PartialStepRight(handler, sequence[i], sequence[i + 1]);
434             }
435             if (sequence.Length >= 3)
436             {
437                 StepLeft(handler, sequence[sequence.Length - 2],
438                     ↪ sequence[sequence.Length - 1]);
439             }
440         }
441         return results;
442     });
443 }
444
445 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
446 {

```

```

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     });
485 }
486
487 public const int MaxSequenceFormatSize = 200;
488
489 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
490     ↪ => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
491
492 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
493     ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
494     ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
495     ↪ elementToString, insertComma, knownElements));
496
497 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
498     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
499     ↪ LinkIndex[] knownElements)
500 {
501     var linksInSequence = new HashSet<ulong>(knownElements);
502     //var entered = new HashSet<ulong>();
503     var sb = new StringBuilder();
504     sb.Append('{');
505     if (links.Exists(sequenceLink))
506     {
507         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
508             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
509             ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
510         {
511             if (insertComma && sb.Length > 1)
512             {
513                 sb.Append(',');
514             }
515             //if (entered.Contains(element))
516             //{
517             //    sb.Append('{');
518             //    elementToString(sb, element);
519             //    sb.Append('}');
520             //}
521             //else
522             elementToString(sb, element);
523         }
524     }
525     sb.Append('}');
526     return sb.ToString();
527 }

```

```

514         if (sb.Length < MaxSequenceFormatSize)
515         {
516             return true;
517         }
518         sb.Append(insertComma ? ", ..." : "...");
519         return false;
520     });
521 }
522 sb.Append('}');
523 return sb.ToString();
524 }
525
526 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
    ↪ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
    ↪ knownElements);
527
528 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
    ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    ↪ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
    ↪ sequenceLink, elementToString, insertComma, knownElements));
529
530 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
    ↪ LinkIndex[] knownElements)
531 {
532     var linksInSequence = new HashSet<ulong>(knownElements);
533     var entered = new HashSet<ulong>();
534     var sb = new StringBuilder();
535     sb.Append('{');
536     if (links.Exists(sequenceLink))
537     {
538         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
539             x => linksInSequence.Contains(x) || links.IsFullPoint(x),
540             ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
541             {
542                 if (insertComma && sb.Length > 1)
543                 {
544                     sb.Append(',');
545                 }
546                 if (entered.Contains(element))
547                 {
548                     sb.Append('{');
549                     elementToString(sb, element);
550                     sb.Append('}');
551                 }
552                 else
553                 {
554                     elementToString(sb, element);
555                 }
556                 if (sb.Length < MaxSequenceFormatSize)
557                 {
558                     return true;
559                 }
560                 sb.Append(insertComma ? ", ..." : "...");
561                 return false;
562             });
563     }
564     sb.Append('}');
565     return sb.ToString();
566 }
567
568 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
569 {
570     return _sync.ExecuteReadOperation(() =>
571     {
572         if (sequence.Length > 0)
573         {
574             Links.EnsureLinkExists(sequence);
575             var results = new HashSet<ulong>();
576             for (var i = 0; i < sequence.Length; i++)
577             {
578                 AllUsagesCore(sequence[i], results);
579             }
580             var filteredResults = new List<ulong>();
581             var linksInSequence = new HashSet<ulong>(sequence);
582             foreach (var result in results)
583             {
584                 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         if (filterPosition == (sequence.Length - 1))
614         {
615             filteredResults.Add(result);
616         }
617     }
618     return filteredResults;
619 }
620 return new List<ulong>();
621 });
622 }
623
624 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
625 {
626     return _sync.ExecuteReadOperation(() =>
627     {
628         if (sequence.Length > 0)
629         {
630             Links.EnsureLinkExists(sequence);
631             var results = new HashSet<ulong>();
632             for (var i = 0; i < sequence.Length; i++)
633             {
634                 AllUsagesCore(sequence[i], results);
635             }
636             var filteredResults = new HashSet<ulong>();
637             var matcher = new Matcher(this, sequence, filteredResults, null);
638             matcher.AddAllPartialMatchedToResults(results);
639             return filteredResults;
640         }
641         return new HashSet<ulong>();
642     });
643 }
644
645 public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
646 ↪ params ulong[] sequence)
647 {
648     return _sync.ExecuteReadOperation(() =>
649     {
650         if (sequence.Length > 0)
651         {
652             Links.EnsureLinkExists(sequence);
653
654             var results = new HashSet<ulong>();
655             var filteredResults = new HashSet<ulong>();
656             var matcher = new Matcher(this, sequence, filteredResults, handler);
657             for (var i = 0; i < sequence.Length; i++)
658             {
659                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
660                 {
661                     return false;
662                 }
663             }
664         }
665     });
666 }

```

```

660         }
661         return true;
662     }
663     return true;
664 });
665 }
666
667 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
668 //{
669 //    return Sync.ExecuteReadOperation(() =>
670 //    {
671 //        if (sequence.Length > 0)
672 //        {
673 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
674 //
675 //            var firstResults = new HashSet<ulong>();
676 //            var lastResults = new HashSet<ulong>();
677 //
678 //            var first = sequence.First(x => x != LinksConstants.Any);
679 //            var last = sequence.Last(x => x != LinksConstants.Any);
680 //
681 //            AllUsagesCore(first, firstResults);
682 //            AllUsagesCore(last, lastResults);
683 //
684 //            firstResults.IntersectWith(lastResults);
685 //
686 //            //for (var i = 0; i < sequence.Length; i++)
687 //            //    AllUsagesCore(sequence[i], results);
688 //
689 //            var filteredResults = new HashSet<ulong>();
690 //            var matcher = new Matcher(this, sequence, filteredResults, null);
691 //            matcher.AddAllPartialMatchedToResults(firstResults);
692 //            return filteredResults;
693 //        }
694 //
695 //        return new HashSet<ulong>();
696 //    });
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
725 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
726     ↪ IList<ulong> sequence)
727 {
728     return _sync.ExecuteReadOperation(() =>
729     {
730         if (sequence.Count > 0)
731         {
732             Links.EnsureLinkExists(sequence);
733             var results = new HashSet<LinkIndex>();
734             //var nextResults = new HashSet<ulong>();
735             //for (var i = 0; i < sequence.Length; i++)
736             //{
737                 AllUsagesCore(sequence[i], nextResults);
738                 if (results.IsNullOrEmpty())

```

```

737         // {
738         //     results = nextResults;
739         //     nextResults = new HashSet<ulong>();
740         // }
741         // else
742         // {
743         //     results.IntersectWith(nextResults);
744         //     nextResults.Clear();
745         // }
746     //}
747     var collector1 = new AllUsagesCollector1(Links.Unsync, results);
748     collector1.Collect(Links.Unsync.GetLink(sequence[0]));
749     var next = new HashSet<ulong>();
750     for (var i = 1; i < sequence.Count; i++)
751     {
752         var collector = new AllUsagesCollector1(Links.Unsync, next);
753         collector.Collect(Links.Unsync.GetLink(sequence[i]));
754
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 }
766
767 // Does not work
768 //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
769     ↪ params ulong[] sequence)
770 //{
771 //    var visited = new HashSet<ulong>();
772 //    var results = new HashSet<ulong>();
773 //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
774     ↪ true; }, readAsElements);
775 //    var last = sequence.Length - 1;
776 //    for (var i = 0; i < last; i++)
777 //    {
778 //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
779 //    }
780 //    return results;
781 //}
782
783 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
784 {
785     return _sync.ExecuteReadOperation(() =>
786     {
787         if (sequence.Length > 0)
788         {
789             Links.EnsureLinkExists(sequence);
790             //var firstElement = sequence[0];
791             //if (sequence.Length == 1)
792             //{
793                 //results.Add(firstElement);
794                 //return results;
795             //}
796             //if (sequence.Length == 2)
797             //{
798                 //var doublet = _links.SearchCore(firstElement, sequence[1]);
799                 //if (doublet != Doublets.Links.Null)
800                 //    results.Add(doublet);
801                 //return results;
802             //}
803             //var lastElement = sequence[sequence.Length - 1];
804             //Func<ulong, bool> handler = x =>
805             //{
806                 //if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
807                 //    ↪ results.Add(x);
808                 //return true;
809             //};
810             //if (sequence.Length >= 2)
811                 StepRight(handler, sequence[0], sequence[1]);

```

```

809 //var last = sequence.Length - 2;
810 //for (var i = 1; i < last; i++)
811 //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
812 //if (sequence.Length >= 3)
813 //    StepLeft(handler, sequence[sequence.Length - 2],
814 //        ↪ sequence[sequence.Length - 1]);
815 ///////if (sequence.Length == 1)
816 ///////
817 ///////
818 ///////throw new NotImplementedException(); // all sequences, containing
819 //        ↪ this element?
820 ///////
821 ///////if (sequence.Length == 2)
822 ///////
823 ///////
824 ///////var results = new List<ulong>();
825 ///////PartialStepRight(results.Add, sequence[0], sequence[1]);
826 ///////return results;
827 ///////
828 ///////var matches = new List<List<ulong>>();
829 ///////var last = sequence.Length - 1;
830 ///////for (var i = 0; i < last; i++)
831 ///////
832 ///////
833 ///////
834 ///////var results = new List<ulong>();
835 /////////StepRight(results.Add, sequence[i], sequence[i + 1]);
836 ///////PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
837 ///////if (results.Count > 0)
838 ///////    matches.Add(results);
839 ///////else
840 ///////    return results;
841 ///////if (matches.Count == 2)
842 ///////
843 ///////
844 ///////var merged = new List<ulong>();
845 ///////for (var j = 0; j < matches[0].Count; j++)
846 ///////    for (var k = 0; k < matches[1].Count; k++)
847 ///////        CloseInnerConnections(merged.Add, matches[0][j],
848 //            ↪ matches[1][k]);
849 ///////if (merged.Count > 0)
850 ///////    matches = new List<List<ulong>> { merged };
851 ///////else
852 ///////    return new List<ulong>();
853 ///////
854 ///////
855 ///////if (matches.Count > 0)
856 ///////
857 ///////
858 ///////var usages = new HashSet<ulong>();
859 ///////for (int i = 0; i < sequence.Length; i++)
860 ///////
861 ///////
862 ///////    AllUsagesCore(sequence[i], usages);
863 ///////
864 ///////
865 ///////for (int i = 0; i < matches[0].Count; i++)
866 ///////    //    AllUsagesCore(matches[0][i], usages);
867 ///////    //usages.UnionWith(matches[0]);
868 ///////    return usages.ToList();
869 ///////
870 //var firstLinkUsages = new HashSet<ulong>();
871 AllUsagesCore(sequence[0], firstLinkUsages);
872 firstLinkUsages.Add(sequence[0]);
873 //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
874 //    ↪ sequence[0] }; // or all sequences, containing this element?
875 //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
876 //    ↪ 1).ToList();
877 var results = new HashSet<ulong>();
878 foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
879     ↪ firstLinkUsages, 1))
880 {
881     AllUsagesCore(match, results);
882 }
883 return results.ToList();
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 }

```

```

880     return _sync.ExecuteReadOperation(() =>
881     {
882         var usages = new HashSet<ulong>();
883         AllUsagesCore(link, usages);
884         return usages;
885     });
886 }
887
888 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
889 // той связи с которой начинался поиск (STTTSSSTT),
890 // причём достаточно одного бита для хранения перехода влево или вправо
891 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
892 {
893     bool handler(ulong doublet)
894     {
895         if (usages.Add(doublet))
896         {
897             AllUsagesCore(doublet, usages);
898         }
899         return true;
900     }
901     Links.Unsync.Each(link, Constants.Any, handler);
902     Links.Unsync.Each(Constants.Any, link, handler);
903 }
904
905 public HashSet<ulong> AllBottomUsages(ulong link)
906 {
907     return _sync.ExecuteReadOperation(() =>
908     {
909         var visits = new HashSet<ulong>();
910         var usages = new HashSet<ulong>();
911         AllBottomUsagesCore(link, visits, usages);
912         return usages;
913     });
914 }
915
916 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
917     usages)
918 {
919     bool handler(ulong doublet)
920     {
921         if (visits.Add(doublet))
922         {
923             AllBottomUsagesCore(doublet, visits, usages);
924         }
925         return true;
926     }
927     if (Links.Unsync.Count(Constants.Any, link) == 0)
928     {
929         usages.Add(link);
930     }
931     else
932     {
933         Links.Unsync.Each(link, Constants.Any, handler);
934         Links.Unsync.Each(Constants.Any, link, handler);
935     }
936 }
937
938 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
939 {
940     if (Options.UseSequenceMarker)
941     {
942         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
943             Options.MarkedSequenceMatcher, symbol);
944         return counter.Count();
945     }
946     else
947     {
948         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
949             symbol);
950         return counter.Count();
951     }
952 }
953
954 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
955     LinkIndex> outerHandler)
956 {
957     bool handler(ulong doublet)

```



```

953     {
954         if (usages.Add(douplet))
955         {
956             if (outerHandler(new LinkAddress<LinkIndex>(douplet)) != Constants.Continue)
957             {
958                 return false;
959             }
960             if (!AllUsagesCore1(douplet, usages, outerHandler))
961             {
962                 return false;
963             }
964         }
965         return true;
966     }
967     return Links.Unsync.Each(link, Constants.Any, handler)
968         && Links.Unsync.Each(Constants.Any, link, handler);
969 }
970
971 public void CalculateAllUsages(ulong[] totals)
972 {
973     var calculator = new AllUsagesCalculator(Links, totals);
974     calculator.Calculate();
975 }
976
977 public void CalculateAllUsages2(ulong[] totals)
978 {
979     var calculator = new AllUsagesCalculator2(Links, totals);
980     calculator.Calculate();
981 }
982
983 private class AllUsagesCalculator
984 {
985     private readonly SynchronizedLinks<ulong> _links;
986     private readonly ulong[] _totals;
987
988     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
989     {
990         _links = links;
991         _totals = totals;
992     }
993
994     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
995         ↪ CalculateCore);
996
997     private bool CalculateCore(ulong link)
998     {
999         if (_totals[link] == 0)
1000         {
1001             var total = 1UL;
1002             _totals[link] = total;
1003             var visitedChildren = new HashSet<ulong>();
1004             bool linkCalculator(ulong child)
1005             {
1006                 if (link != child && visitedChildren.Add(child))
1007                 {
1008                     total += _totals[child] == 0 ? 1 : _totals[child];
1009                 }
1010                 return true;
1011             }
1012             _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1013             _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1014             _totals[link] = total;
1015         }
1016         return true;
1017     }
1018 }
1019
1020 private class AllUsagesCalculator2
1021 {
1022     private readonly SynchronizedLinks<ulong> _links;
1023     private readonly ulong[] _totals;
1024
1025     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1026     {
1027         _links = links;
1028         _totals = totals;
1029     }
1030
1031     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1032         ↪ CalculateCore);

```

```

1031 private bool IsElement(ulong link)
1032 {
1033     // _linksInSequence.Contains(link) ||
1034     return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1035         ↪ link;
1036 }
1037
1038 private bool CalculateCore(ulong link)
1039 {
1040     // TODO: Проработать защиту от заикливания
1041     // Основано на SequenceWalker.WalkLeft
1042     Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1043     Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1044     Func<ulong, bool> isElement = IsElement;
1045     void visitLeaf(ulong parent)
1046     {
1047         if (link != parent)
1048         {
1049             _totals[parent]++;
1050         }
1051     }
1052     void visitNode(ulong parent)
1053     {
1054         if (link != parent)
1055         {
1056             _totals[parent]++;
1057         }
1058     }
1059     var stack = new Stack();
1060     var element = link;
1061     if (isElement(element))
1062     {
1063         visitLeaf(element);
1064     }
1065     else
1066     {
1067         while (true)
1068         {
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 private class AllUsagesCollector
1102 {
1103     private readonly ILinks<ulong> _links;
1104     private readonly HashSet<ulong> _usages;
1105
1106     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1107     {
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 заикливания
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
1201     ↪ right)
1202 {
1203     TryStepLeftUp(handler, left, right);
1204     TryStepRightUp(handler, right, left);
1205 }
1206
1207 private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
1208     ↪ right)
1209 {
1210     // Direct
1211     if (left == right)
1212     {
1213         handler(new LinkAddress<LinkIndex>(left));
1214     }
1215     var doublet = Links.Unsync.SearchOrDefault(left, right);
1216     if (doublet != Constants.Null)
1217     {
1218         handler(new LinkAddress<LinkIndex>(doublet));
1219     }
1220     // Inner
1221     CloseInnerConnections(handler, left, right);
1222     // Outer
1223     StepLeft(handler, left, right);
1224     StepRight(handler, left, right);
1225     PartialStepRight(handler, left, right);
1226     PartialStepLeft(handler, left, right);
1227 }
1228
1229 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
1230     ↪ HashSet<ulong> previousMatchings, long startAt)
1231 {
1232     if (startAt >= sequence.Length) // ?
1233     {
1234         return previousMatchings;
1235     }
1236     var secondLinkUsages = new HashSet<ulong>();
1237     AllUsagesCore(sequence[startAt], secondLinkUsages);
1238     secondLinkUsages.Add(sequence[startAt]);
1239     var matchings = new HashSet<ulong>();
1240     var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
1241     //for (var i = 0; i < previousMatchings.Count; i++)
1242     foreach (var secondLinkUsage in secondLinkUsages)
1243     {
1244         foreach (var previousMatching in previousMatchings)
1245         {
1246             //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
1247             ↪ secondLinkUsage);
1248             StepRight(filler.AddFirstAndReturnConstant, previousMatching,
1249             ↪ secondLinkUsage);
1250             TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
1251             ↪ previousMatching);
1252             //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1253             ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1254             ↪ желаемым результатам.
1255             PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
1256             ↪ secondLinkUsage);
1257         }
1258     }
1259     if (matchings.Count == 0)
1260     {
1261         return matchings;
1262     }
1263     return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1264 }

```

```

1256 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
1257     ↳ links, params ulong[] sequence)
1258 {
1259     if (sequence == null)
1260     {
1261         return;
1262     }
1263     for (var i = 0; i < sequence.Length; i++)
1264     {
1265         if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
1266             ↳ !links.Exists(sequence[i]))
1267         {
1268             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1269                 ↳ $"patternSequence[{i}]");
1270         }
1271     }
1272 }
1273 // Pattern Matching -> Key To Triggers
1274 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1275 {
1276     return _sync.ExecuteReadOperation(() =>
1277     {
1278         patternSequence = Simplify(patternSequence);
1279         if (patternSequence.Length > 0)
1280         {
1281             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1282             var uniqueSequenceElements = new HashSet<ulong>();
1283             for (var i = 0; i < patternSequence.Length; i++)
1284             {
1285                 if (patternSequence[i] != Constants.Any && patternSequence[i] !=
1286                     ↳ ZeroOrMany)
1287                 {
1288                     uniqueSequenceElements.Add(patternSequence[i]);
1289                 }
1290             }
1291             var results = new HashSet<ulong>();
1292             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1293             {
1294                 AllUsagesCore(uniqueSequenceElement, results);
1295             }
1296             var filteredResults = new HashSet<ulong>();
1297             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1298             matcher.AddAllPatternMatchedToResults(results);
1299             return filteredResults;
1300         }
1301         return new HashSet<ulong>();
1302     });
1303 }
1304 // Найти все возможные связи между указанным списком связей.
1305 // Находит связи между всеми указанными связями в любом порядке.
1306 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
1307 ↳ несколько раз в последовательности)
1308 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1309 {
1310     return _sync.ExecuteReadOperation(() =>
1311     {
1312         var results = new HashSet<ulong>();
1313         if (linksToConnect.Length > 0)
1314         {
1315             Links.EnsureLinkExists(linksToConnect);
1316             AllUsagesCore(linksToConnect[0], results);
1317             for (var i = 1; i < linksToConnect.Length; i++)
1318             {
1319                 var next = new HashSet<ulong>();
1320                 AllUsagesCore(linksToConnect[i], next);
1321                 results.IntersectWith(next);
1322             }
1323             return results;
1324         }
1325     });
1326 }
1327 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1328 {
1329     return _sync.ExecuteReadOperation(() =>

```

```

1329     {
1330         var results = new HashSet<ulong>();
1331         if (linksToConnect.Length > 0)
1332         {
1333             Links.EnsureLinkExists(linksToConnect);
1334             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1335             collector1.Collect(linksToConnect[0]);
1336             var next = new HashSet<ulong>();
1337             for (var i = 1; i < linksToConnect.Length; i++)
1338             {
1339                 var collector = new AllUsagesCollector(Links.Unsync, next);
1340                 collector.Collect(linksToConnect[i]);
1341                 results.IntersectWith(next);
1342                 next.Clear();
1343             }
1344             return results;
1345         });
1346     });
1347 }
1348
1349 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1350 {
1351     return _sync.ExecuteReadOperation(() =>
1352     {
1353         var results = new HashSet<ulong>();
1354         if (linksToConnect.Length > 0)
1355         {
1356             Links.EnsureLinkExists(linksToConnect);
1357             var collector1 = new AllUsagesCollector(Links, results);
1358             collector1.Collect(linksToConnect[0]);
1359             //AllUsagesCore(linksToConnect[0], results);
1360             for (var i = 1; i < linksToConnect.Length; i++)
1361             {
1362                 var next = new HashSet<ulong>();
1363                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1364                 collector.Collect(linksToConnect[i]);
1365                 //AllUsagesCore(linksToConnect[i], next);
1366                 //results.IntersectWith(next);
1367                 results = next;
1368             }
1369             return results;
1370         });
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         {

```

```

1405         if (zeroOrManyStepped)
1406         {
1407             continue;
1408         }
1409         zeroOrManyStepped = true;
1410     }
1411     else
1412     {
1413         //if (zeroOrManyStepped) Is it efficient?
1414         zeroOrManyStepped = false;
1415     }
1416     newLength++;
1417 }
1418 // Строим новую последовательность
1419 zeroOrManyStepped = false;
1420 var newSequence = new ulong[newLength];
1421 long j = 0;
1422 for (var i = 0; i < sequence.Length; i++)
1423 {
1424     //var current = zeroOrManyStepped;
1425     //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1426     //if (current && zeroOrManyStepped)
1427     //    continue;
1428     //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1429     //if (zeroOrManyStepped && newZeroOrManyStepped)
1430     //    continue;
1431     //zeroOrManyStepped = newZeroOrManyStepped;
1432     if (sequence[i] == ZeroOrMany)
1433     {
1434         if (zeroOrManyStepped)
1435         {
1436             continue;
1437         }
1438         zeroOrManyStepped = true;
1439     }
1440     else
1441     {
1442         //if (zeroOrManyStepped) Is it efficient?
1443         zeroOrManyStepped = false;
1444     }
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,
1453     ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1454     var simplifiedSequence = Simplify(sequence);
1455 }
1456
1457 public List<ulong> GetSimilarSequences() => new List<ulong>();
1458
1459 public void Prediction()
1460 {
1461     //_links
1462     //sequences
1463 }
1464
1465 #region From Triplets
1466
1467 //public static void DeleteSequence(Link sequence)
1468 //{
1469 //}
1470
1471 public List<ulong> CollectMatchingSequences(ulong[] links)
1472 {
1473     if (links.Length == 1)
1474     {
1475         throw new Exception("Подпоследовательности с одним элементом не
1476         ↪ поддерживаются.");
1477     }
1478     var leftBound = 0;
1479     var rightBound = links.Length - 1;
1480     var left = links[leftBound++];
1481     var right = links[rightBound--];
1482     var results = new List<ulong>();
1483     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);

```

```

1482         return results;
1483     }
1484
1485     private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1486     ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1487     {
1488         var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1489         var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1490         if (leftLinkTotalReferers <= rightLinkTotalReferers)
1491         {
1492             var nextLeftLink = middleLinks[leftBound];
1493             var elements = GetRightElements(leftLink, nextLeftLink);
1494             if (leftBound <= rightBound)
1495             {
1496                 for (var i = elements.Length - 1; i >= 0; i--)
1497                 {
1498                     var element = elements[i];
1499                     if (element != 0)
1500                     {
1501                         CollectMatchingSequences(element, leftBound + 1, middleLinks,
1502                         ↪ rightLink, rightBound, ref results);
1503                     }
1504                 }
1505             }
1506             else
1507             {
1508                 for (var i = elements.Length - 1; i >= 0; i--)
1509                 {
1510                     var element = elements[i];
1511                     if (element != 0)
1512                     {
1513                         results.Add(element);
1514                     }
1515                 }
1516             }
1517         }
1518         else
1519         {
1520             var nextRightLink = middleLinks[rightBound];
1521             var elements = GetLeftElements(rightLink, nextRightLink);
1522             if (leftBound <= rightBound)
1523             {
1524                 for (var i = elements.Length - 1; i >= 0; i--)
1525                 {
1526                     var element = elements[i];
1527                     if (element != 0)
1528                     {
1529                         CollectMatchingSequences(leftLink, leftBound, middleLinks,
1530                         ↪ elements[i], rightBound - 1, ref results);
1531                     }
1532                 }
1533             }
1534             else
1535             {
1536                 for (var i = elements.Length - 1; i >= 0; i--)
1537                 {
1538                     var element = elements[i];
1539                     if (element != 0)
1540                     {
1541                         results.Add(element);
1542                     }
1543                 }
1544             }
1545         }
1546     }
1547
1548     public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1549     {
1550         var result = new ulong[5];
1551         TryStepRight(startLink, rightLink, result, 0);
1552         Links.Each(Constants.Any, startLink, couple =>
1553         {
1554             if (couple != startLink)
1555             {
1556                 if (TryStepRight(couple, rightLink, result, 2))
1557                 {
1558                     return false;
1559                 }
1560             }
1561         }
1562     }

```



```

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     });
1611     if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1612     {
1613         result[4] = leftLink;
1614     }
1615     return result;
1616 }
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         }
1636     });

```

```

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         return true;
1645     });
1646     return added > 0;
1647 }
1648
1649 #endregion
1650
1651 #region Walkers
1652
1653 public class PatternMatcher : RightSequenceWalker<ulong>
1654 {
1655     private readonly Sequences _sequences;
1656     private readonly ulong[] _patternSequence;
1657     private readonly HashSet<LinkIndex> _linksInSequence;
1658     private readonly HashSet<LinkIndex> _results;
1659
1660     #region Pattern Match
1661
1662     enum PatternBlockType
1663     {
1664         Undefined,
1665         Gap,
1666         Elements
1667     }
1668
1669     struct PatternBlock
1670     {
1671         public PatternBlockType Type;
1672         public long Start;
1673         public long Stop;
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>();

```

```

1711 var patternBlock = new PatternBlock();
1712 for (var i = 0; i < _patternSequence.Length; i++)
1713 {
1714     if (patternBlock.Type == PatternBlockType.Undefined)
1715     {
1716         if (_patternSequence[i] == _sequences.Constants.Any)
1717         {
1718             patternBlock.Type = PatternBlockType.Gap;
1719             patternBlock.Start = 1;
1720             patternBlock.Stop = 1;
1721         }
1722         else if (_patternSequence[i] == ZeroOrMany)
1723         {
1724             patternBlock.Type = PatternBlockType.Gap;
1725             patternBlock.Start = 0;
1726             patternBlock.Stop = long.MaxValue;
1727         }
1728         else
1729         {
1730             patternBlock.Type = PatternBlockType.Elements;
1731             patternBlock.Start = i;
1732             patternBlock.Stop = i;
1733         }
1734     }
1735     else if (patternBlock.Type == PatternBlockType.Elements)
1736     {
1737         if (_patternSequence[i] == _sequences.Constants.Any)
1738         {
1739             pattern.Add(patternBlock);
1740             patternBlock = new PatternBlock
1741             {
1742                 Type = PatternBlockType.Gap,
1743                 Start = 1,
1744                 Stop = 1
1745             };
1746         }
1747         else if (_patternSequence[i] == ZeroOrMany)
1748         {
1749             pattern.Add(patternBlock);
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
1830 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
1831 //    ↪ long maximumGap)
1832 //{
1833 //    mininumGap = 0;
1834 //    maximumGap = 0;
1835 //    element = 0;
1836 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1837 //    {
1838 //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1839 //            mininumGap++;
1840 //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1841 //            maximumGap = long.MaxValue;
1842 //        else
1843 //            break;
1844 //    }
1845
1846 //    if (maximumGap < mininumGap)
1847 //        maximumGap = mininumGap;
1848 //}
1849
1850 private bool PatternMatchCore(LinkIndex element)
1851 {
1852     if (_patternPosition >= _pattern.Count)
1853     {
1854         _patternPosition = -2;
1855         return false;
1856     }
1857     var currentPatternBlock = _pattern[_patternPosition];
1858     if (currentPatternBlock.Type == PatternBlockType.Gap)
1859     {
1860         //var currentMatchingBlockLength = (_sequencePosition -
1861         ↪ _lastMatchedBlockPosition);
1862         if (_sequencePosition < currentPatternBlock.Start)
1863         {
1864             _sequencePosition++;
1865             return true; // Двигаемся дальше
1866         }
1867         // Это последний блок
1868         if (_pattern.Count == _patternPosition + 1)
1869         {

```

```

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
1959     #endregion
1960 }
1961 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections;
6  using Platform.Collections.Lists;
7  using Platform.Collections.Stacks;
8  using Platform.Threading.Synchronization;
9  using Platform.Data.Doublets.Sequences.Walkers;
10 using LinkIndex = System.UInt64;
11
12 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     /// <summary>
17     /// Представляет коллекцию последовательностей связей.
18     /// </summary>
19     /// <remarks>
20     /// Обязательно реализовать атомарность каждого публичного метода.
21     ///
22     /// TODO:
23     ///
24     /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
25     /// через естественную группировку по unicode типам, все whitespace вместе, все символы
26     /// ↪ вместе, все числа вместе и т.п.
27     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
28     /// ↪ графа)
29     ///
30     /// х*у - найти все связи между, в последовательностях любой формы, если не стоит
31     /// ↪ ограничитель на то, что является последовательностью, а что нет,
32     /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
33     /// ↪ порядке.
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     ///
52     /// Блокчейн и/или гит для распределённой записи транзакций.
53     ///
54     /// </remarks>
55     public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
56     ↪ (после завершения реализации Sequences)
57     {
58         /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
59         ↪ связей.</summary>
60         public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
61     }
62 }

```

```

55 public SequencesOptions<LinkIndex> Options { get; }
56 public SynchronizedLinks<LinkIndex> Links { get; }
57 private readonly ISynchronization _sync;
58
59 public LinksConstants<LinkIndex> Constants { get; }
60
61 public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
62 {
63     Links = links;
64     _sync = links.SyncRoot;
65     Options = options;
66     Options.ValidateOptions();
67     Options.InitOptions(Links);
68     Constants = links.Constants;
69 }
70
71 public Sequences(SynchronizedLinks<LinkIndex> links)
72 : this(links, new SequencesOptions<LinkIndex>())
73 {
74 }
75
76 public bool IsSequence(LinkIndex sequence)
77 {
78     return _sync.ExecuteReadOperation(() =>
79     {
80         if (Options.UseSequenceMarker)
81         {
82             return Options.MarkedSequenceMatcher.IsMatched(sequence);
83         }
84         return !Links.Unsync.IsPartialPoint(sequence);
85     });
86 }
87
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 private LinkIndex GetSequenceByElements(LinkIndex sequence)
90 {
91     if (Options.UseSequenceMarker)
92     {
93         return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
94     }
95     return sequence;
96 }
97
98 private LinkIndex GetSequenceElements(LinkIndex sequence)
99 {
100     if (Options.UseSequenceMarker)
101     {
102         var linkContents = new Link<ulong>(Links.GetLink(sequence));
103         if (linkContents.Source == Options.SequenceMarkerLink)
104         {
105             return linkContents.Target;
106         }
107         if (linkContents.Target == Options.SequenceMarkerLink)
108         {
109             return linkContents.Source;
110         }
111     }
112     return sequence;
113 }
114
115 #region Count
116
117 public LinkIndex Count(ICollection<LinkIndex> restrictions)
118 {
119     if (restrictions.IsNullOrEmpty())
120     {
121         return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
122     }
123     if (restrictions.Count == 1) // Первая связь это адрес
124     {
125         var sequenceIndex = restrictions[0];
126         if (sequenceIndex == Constants.Null)
127         {
128             return 0;
129         }
130         if (sequenceIndex == Constants.Any)
131         {
132             return Count(null);
133         }
134     }
135 }

```

```

134         if (Options.UseSequenceMarker)
135         {
136             return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
137         }
138         return Links.Exists(sequenceIndex) ? 1UL : 0;
139     }
140     throw new NotImplementedException();
141 }
142
143 private LinkIndex CountUsages(params LinkIndex[] restrictions)
144 {
145     if (restrictions.Length == 0)
146     {
147         return 0;
148     }
149     if (restrictions.Length == 1) // Первая связь это адрес
150     {
151         if (restrictions[0] == Constants.Null)
152         {
153             return 0;
154         }
155         var any = Constants.Any;
156         if (Options.UseSequenceMarker)
157         {
158             var elementsLink = GetSequenceElements(restrictions[0]);
159             var sequenceLink = GetSequenceByElements(elementsLink);
160             if (sequenceLink != Constants.Null)
161             {
162                 return Links.Count(any, sequenceLink) + Links.Count(any, elementsLink) -
163                     ↪ 1;
164             }
165             return Links.Count(any, elementsLink);
166         }
167         return Links.Count(any, restrictions[0]);
168     }
169     throw new NotImplementedException();
170 }
171 #endregion
172
173 #region Create
174
175 public LinkIndex Create(IList<LinkIndex> restrictions)
176 {
177     return _sync.ExecuteWriteOperation(() =>
178     {
179         if (restrictions.IsNullOrEmpty())
180         {
181             return Constants.Null;
182         }
183         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
184         return CreateCore(restrictions);
185     });
186 }
187
188 private LinkIndex CreateCore(IList<LinkIndex> restrictions)
189 {
190     LinkIndex[] sequence = restrictions.SkipFirst();
191     if (Options.UseIndex)
192     {
193         Options.Index.Add(sequence);
194     }
195     var sequenceRoot = default(LinkIndex);
196     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
197     {
198         var matches = Each(restrictions);
199         if (matches.Count > 0)
200         {
201             sequenceRoot = matches[0];
202         }
203     }
204     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
205     {
206         return CompactCore(sequence);
207     }
208     if (sequenceRoot == default)
209     {
210         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);

```



```

211     }
212     if (Options.UseSequenceMarker)
213     {
214         return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
215     }
216     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
217 }
218
219 #endregion
220
221 #region Each
222
223 public List<LinkIndex> Each(IList<LinkIndex> sequence)
224 {
225     var results = new List<LinkIndex>();
226     var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
227     Each(filler.AddFirstAndReturnConstant, sequence);
228     return results;
229 }
230
231 public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
    ↪ restrictions)
232 {
233     return _sync.ExecuteReadOperation(() =>
234     {
235         if (restrictions.IsNullOrEmpty())
236         {
237             return Constants.Continue;
238         }
239         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
240         if (restrictions.Count == 1)
241         {
242             var link = restrictions[0];
243             var any = Constants.Any;
244             if (link == any)
245             {
246                 if (Options.UseSequenceMarker)
247                 {
248                     return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
    ↪ Options.SequenceMarkerLink, any));
249                 }
250                 else
251                 {
252                     return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
    ↪ any));
253                 }
254             }
255             if (Options.UseSequenceMarker)
256             {
257                 var sequenceLinkValues = Links.Unsync.GetLink(link);
258                 if (sequenceLinkValues[Constants.SourcePart] ==
    ↪ Options.SequenceMarkerLink)
259                 {
260                     link = sequenceLinkValues[Constants.TargetPart];
261                 }
262             }
263             var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
264             sequence[0] = link;
265             return handler(sequence);
266         }
267         else if (restrictions.Count == 2)
268         {
269             throw new NotImplementedException();
270         }
271         else if (restrictions.Count == 3)
272         {
273             return Links.Unsync.Each(handler, restrictions);
274         }
275         else
276         {
277             var sequence = restrictions.SkipFirst();
278             if (Options.UseIndex && !Options.Index.MightContain(sequence))
279             {
280                 return Constants.Break;
281             }
282             return EachCore(handler, sequence);
283         }
284     });

```

```

285 }
286
287 private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
    ↪ values)
288 {
289     var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
290     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
    ↪ Id.
291     Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
    ↪ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
    ↪ matcher.HandleFullMatched;
292     //if (sequence.Length >= 2)
293     if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
294     {
295         return Constants.Break;
296     }
297     var last = values.Count - 2;
298     for (var i = 1; i < last; i++)
299     {
300         if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
    ↪ Constants.Continue)
301         {
302             return Constants.Break;
303         }
304     }
305     if (values.Count >= 3)
306     {
307         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
    ↪ != Constants.Continue)
308         {
309             return Constants.Break;
310         }
311     }
312     return Constants.Continue;
313 }
314
315 private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↪ left, LinkIndex right)
316 {
317     return Links.Unsync.Each(doublet =>
318     {
319         var doubletIndex = doublet[Constants.IndexPart];
320         if (StepRight(handler, doubletIndex, right) != Constants.Continue)
321         {
322             return Constants.Break;
323         }
324         if (left != doubletIndex)
325         {
326             return PartialStepRight(handler, doubletIndex, right);
327         }
328         return Constants.Continue;
329     }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
330 }
331
332 private LinkIndex StepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↪ LinkIndex right) => Links.Unsync.Each(rightStep => TryStepRightUp(handler, right,
    ↪ rightStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, left,
    ↪ Constants.Any));
333
334 private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↪ right, LinkIndex stepFrom)
335 {
336     var upStep = stepFrom;
337     var firstSource = Links.Unsync.GetTarget(upStep);
338     while (firstSource != right && firstSource != upStep)
339     {
340         upStep = firstSource;
341         firstSource = Links.Unsync.GetSource(upStep);
342     }
343     if (firstSource == right)
344     {
345         return handler(new LinkAddress<LinkIndex>(stepFrom));
346     }
347     return Constants.Continue;
348 }
349

```

```

350 private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↳ LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
    ↳ leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
    ↳ right));
351
352 private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ left, LinkIndex stepFrom)
353 {
354     var upStep = stepFrom;
355     var firstTarget = Links.Unsync.GetSource(upStep);
356     while (firstTarget != left && firstTarget != upStep)
357     {
358         upStep = firstTarget;
359         firstTarget = Links.Unsync.GetTarget(upStep);
360     }
361     if (firstTarget == left)
362     {
363         return handler(new LinkAddress<LinkIndex>(stepFrom));
364     }
365     return Constants.Continue;
366 }
367
368 #endregion
369
370 #region Update
371
372 public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
373 {
374     var sequence = restrictions.SkipFirst();
375     var newSequence = substitution.SkipFirst();
376
377     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
378     {
379         return Constants.Null;
380     }
381     if (sequence.IsNullOrEmpty())
382     {
383         return Create(substitution);
384     }
385     if (newSequence.IsNullOrEmpty())
386     {
387         Delete(restrictions);
388         return Constants.Null;
389     }
390     return _sync.ExecuteWriteOperation((Func<ulong>)(() =>
391     {
392         ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
393         Links.EnsureLinkExists(newSequence);
394         return UpdateCore(sequence, newSequence);
395     })));
396 }
397
398 private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence)
399 {
400     LinkIndex bestVariant;
401     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
402         ↳ !sequence.EqualTo(newSequence))
403     {
404         bestVariant = CompactCore(newSequence);
405     }
406     else
407     {
408         bestVariant = CreateCore(newSequence);
409     }
410     // TODO: Check all options only ones before loop execution
411     // Возможно нужно две версии Each, возвращающий фактические последовательности и с
412     ↳ маркером,
413     // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
414     ↳ можно получить имея только фактические последовательности.
415     foreach (var variant in Each(sequence))
416     {
417         if (variant != bestVariant)
418         {
419             UpdateOneCore(variant, bestVariant);
420         }
421     }
    return bestVariant;
}

```

```

422 private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
423 {
424     if (Options.UseGarbageCollection)
425     {
426         var sequenceElements = GetSequenceElements(sequence);
427         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
428         var sequenceLink = GetSequenceByElements(sequenceElements);
429         var newSequenceElements = GetSequenceElements(newSequence);
430         var newSequenceLink = GetSequenceByElements(newSequenceElements);
431         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
432         {
433             if (sequenceLink != Constants.Null)
434             {
435                 Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
436             }
437             Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
438         }
439         ClearGarbage(sequenceElementsContents.Source);
440         ClearGarbage(sequenceElementsContents.Target);
441     }
442     else
443     {
444         if (Options.UseSequenceMarker)
445         {
446             var sequenceElements = GetSequenceElements(sequence);
447             var sequenceLink = GetSequenceByElements(sequenceElements);
448             var newSequenceElements = GetSequenceElements(newSequence);
449             var newSequenceLink = GetSequenceByElements(newSequenceElements);
450             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
451             {
452                 if (sequenceLink != Constants.Null)
453                 {
454                     Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
455                 }
456                 Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
457             }
458         }
459         else
460         {
461             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
462             {
463                 Links.Unsync.MergeAndDelete(sequence, newSequence);
464             }
465         }
466     }
467 }
468
469 #endregion
470
471 #region Delete
472
473 public void Delete(IList<LinkIndex> restrictions)
474 {
475     _sync.ExecuteWriteOperation(() =>
476     {
477         var sequence = restrictions.SkipFirst();
478         // TODO: Check all options only ones before loop execution
479         foreach (var linkToDelete in Each(sequence))
480         {
481             DeleteOneCore(linkToDelete);
482         }
483     });
484 }
485
486 private void DeleteOneCore(LinkIndex link)
487 {
488     if (Options.UseGarbageCollection)
489     {
490         var sequenceElements = GetSequenceElements(link);
491         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
492         var sequenceLink = GetSequenceByElements(sequenceElements);
493         if (Options.UseCascadeDelete || CountUsages(link) == 0)
494         {
495             if (sequenceLink != Constants.Null)
496             {
497                 Links.Unsync.Delete(sequenceLink);
498             }
499             Links.Unsync.Delete(link);

```

```

500     }
501     ClearGarbage(sequenceElementsContents.Source);
502     ClearGarbage(sequenceElementsContents.Target);
503 }
504 else
505 {
506     if (Options.UseSequenceMarker)
507     {
508         var sequenceElements = GetSequenceElements(link);
509         var sequenceLink = GetSequenceByElements(sequenceElements);
510         if (Options.UseCascadeDelete || CountUsages(link) == 0)
511         {
512             if (sequenceLink != Constants.Null)
513             {
514                 Links.Unsync.Delete(sequenceLink);
515             }
516             Links.Unsync.Delete(link);
517         }
518     }
519     else
520     {
521         if (Options.UseCascadeDelete || CountUsages(link) == 0)
522         {
523             Links.Unsync.Delete(link);
524         }
525     }
526 }
527 }
528
529 #endregion
530
531 #region Compactification
532
533 public void CompactAll()
534 {
535     _sync.ExecuteWriteOperation(() =>
536     {
537         var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
538         for (int i = 0; i < sequences.Count; i++)
539         {
540             var sequence = this.ToList(sequences[i]);
541             Compact(sequence.ShiftRight());
542         }
543     });
544 }
545
546 /// <remarks>
547 /// bestVariant можно выбирать по максимальному числу использований,
548 /// но балансированный позволяет гарантировать уникальность (если есть возможность,
549 /// гарантировать его использование в других местах).
550 ///
551 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
552 /// </remarks>
553 public LinkIndex Compact(ICollection<LinkIndex> sequence)
554 {
555     return _sync.ExecuteWriteOperation(() =>
556     {
557         if (sequence.IsNullOrEmpty())
558         {
559             return Constants.Null;
560         }
561         Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
562         return CompactCore(sequence);
563     });
564 }
565
566 [MethodImpl(MethodImplOptions.AggressiveInlining)]
567 private LinkIndex CompactCore(ICollection<LinkIndex> sequence) => UpdateCore(sequence,
568     ↪ sequence);
569
570 #endregion
571
572 #region Garbage Collection
573
574 /// <remarks>
575 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
576     ↪ определить извне или в унаследованном классе
577 /// </remarks>
578 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

577 private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
578     ↳ !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
579
580 private void ClearGarbage(LinkIndex link)
581 {
582     if (IsGarbage(link))
583     {
584         var contents = new Link<ulong>(Links.GetLink(link));
585         Links.Unsync.Delete(link);
586         ClearGarbage(contents.Source);
587         ClearGarbage(contents.Target);
588     }
589 }
590
591 #endregion
592
593 #region Walkers
594
595 public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
596 {
597     return _sync.ExecuteReadOperation(() =>
598     {
599         var links = Links.Unsync;
600         foreach (var part in Options.Walker.Walk(sequence))
601         {
602             if (!handler(part))
603             {
604                 return false;
605             }
606         }
607         return true;
608     });
609 }
610
611 public class Matcher : RightSequenceWalker<LinkIndex>
612 {
613     private readonly Sequences _sequences;
614     private readonly IList<LinkIndex> _patternSequence;
615     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         }
658     }
659 }

```

```

651         return false;
652     }
653     if (_patternSequence[_filterPosition] != Links.Constants.Any
654         && element != _patternSequence[_filterPosition])
655     {
656         _filterPosition = -1;
657         return false; // Начинается/Продолжается иначе
658     }
659     _filterPosition++;
660     return true;
661 }
662
663 public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
664 {
665     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
666     if (FullMatch(sequenceToMatch))
667     {
668         _results.Add(sequenceToMatch);
669     }
670 }
671
672 public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
673 {
674     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
675     if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
676     {
677         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
678     }
679     return Links.Constants.Continue;
680 }
681
682 public LinkIndex HandleFullMatchedSequence(IList<LinkIndex> restrictions)
683 {
684     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
685     var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
686     if (sequence != Links.Constants.Null && FullMatch(sequenceToMatch) &&
687         ↪ _results.Add(sequenceToMatch))
688     {
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])

```

```

729         {
730             _filterPosition = 0;
731         }
732     }
733     return true; // Ищем дальше
734 }
735
736 public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
737 {
738     if (PartialMatch(sequenceToMatch))
739     {
740         _results.Add(sequenceToMatch);
741     }
742 }
743
744 public LinkIndex HandlePartialMatched(ICollection<LinkIndex> restrictions)
745 {
746     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
747     if (PartialMatch(sequenceToMatch))
748     {
749         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
750     }
751     return Links.Constants.Continue;
752 }
753
754 public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)
755 {
756     foreach (var sequenceToMatch in sequencesToMatch)
757     {
758         if (PartialMatch(sequenceToMatch))
759         {
760             _results.Add(sequenceToMatch);
761         }
762     }
763 }
764
765 public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
↪ sequencesToMatch)
766 {
767     foreach (var sequenceToMatch in sequencesToMatch)
768     {
769         if (PartialMatch(sequenceToMatch))
770         {
771             _readAsElements.Add(sequenceToMatch);
772             _results.Add(sequenceToMatch);
773         }
774     }
775 }
776 }
777
778 #endregion
779 }
780 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences
8  {
9      public static class SequencesExtensions
10     {
11         public static TLink Create<TLink>(this ICollection<TLink> sequences, ICollection<TLink[]>
↪ groupedSequence)
12         {
13             var finalSequence = new TLink[groupedSequence.Count];
14             for (var i = 0; i < finalSequence.Length; i++)
15             {
16                 var part = groupedSequence[i];
17                 finalSequence[i] = part.Length == 1 ? part[0] :
↪ sequences.Create(part.ShiftRight());
18             }
19             return sequences.Create(finalSequence.ShiftRight());
20         }
21
22         public static ICollection<TLink> ToList<TLink>(this ICollection<TLink> sequences, TLink sequence)

```



```

23     {
24         var list = new List<TLink>();
25         var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
26         sequences.Each(filler.AddSkipFirstAndReturnConstant, new
            ↪ LinkAddress<TLink>(sequence));
27         return list;
28     }
29 }
30 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Collections.Stacks;
5  using Platform.Converters;
6  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
7  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
8  using Platform.Data.Doublets.Sequences.Converters;
9  using Platform.Data.Doublets.Sequences.Walkers;
10 using Platform.Data.Doublets.Sequences.Indexes;
11 using Platform.Data.Doublets.Sequences.CriterionMatchers;
12
13 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14
15 namespace Platform.Data.Doublets.Sequences
16 {
17     public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
        ↪ ILinks<TLink> must contain GetConstants function.
18     {
19         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
20
21         public TLink SequenceMarkerLink { get; set; }
22         public bool UseCascadeUpdate { get; set; }
23         public bool UseCascadeDelete { get; set; }
24         public bool UseIndex { get; set; } // TODO: Update Index on sequence update/delete.
25         public bool UseSequenceMarker { get; set; }
26         public bool UseCompression { get; set; }
27         public bool UseGarbageCollection { get; set; }
28         public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get; set; }
29         public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }
30
31         public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher { get; set; }
32         public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
33         public ISequenceIndex<TLink> Index { get; set; }
34         public ISequenceWalker<TLink> Walker { get; set; }
35         public bool ReadFullSequence { get; set; }
36
37         // TODO: Реализовать компактификацию при чтении
38         //public bool EnforceSingleSequenceVersionOnRead { get; set; }
39         //public bool UseRequestMarker { get; set; }
40         //public bool StoreRequestResults { get; set; }
41
42         public void InitOptions(ISynchronizedLinks<TLink> links)
43         {
44             if (UseSequenceMarker)
45             {
46                 if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
47                 {
48                     SequenceMarkerLink = links.CreatePoint();
49                 }
50                 else
51                 {
52                     if (!links.Exists(SequenceMarkerLink))
53                     {
54                         var link = links.CreatePoint();
55                         if (!_equalityComparer.Equals(link, SequenceMarkerLink))
56                         {
57                             throw new InvalidOperationException("Cannot recreate sequence marker
                                ↪ link.");
58                         }
59                     }
60                 }
61             }
62             if (MarkedSequenceMatcher == null)
63             {
64                 MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
                    ↪ SequenceMarkerLink);
65             }
66         }
67     }
68 }

```

```

65     }
66     var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
67     if (UseCompression)
68     {
69         if (LinksToSequenceConverter == null)
70         {
71             ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
72             if (UseSequenceMarker)
73             {
74                 totalSequenceSymbolFrequencyCounter = new
75                     ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
76                     ↪ MarkedSequenceMatcher);
77             }
78             else
79             {
80                 totalSequenceSymbolFrequencyCounter = new
81                     ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
82             }
83             var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
84                 ↪ totalSequenceSymbolFrequencyCounter);
85             var compressingConverter = new CompressingConverter<TLink>(links,
86                 ↪ balancedVariantConverter, doubletFrequenciesCache);
87             LinksToSequenceConverter = compressingConverter;
88         }
89     }
90     else
91     {
92         if (LinksToSequenceConverter == null)
93         {
94             LinksToSequenceConverter = balancedVariantConverter;
95         }
96     }
97     if (UseIndex && Index == null)
98     {
99         Index = new SequenceIndex<TLink>(links);
100     }
101     if (Walker == null)
102     {
103         Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
104     }
105 }
106
107 public void ValidateOptions()
108 {
109     if (UseGarbageCollection && !UseSequenceMarker)
110     {
111         throw new NotSupportedException("To use garbage collection UseSequenceMarker
112             ↪ option must be on.");
113     }
114 }

```

1.85 ./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 }

```

1.86 ./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 }

```

1.87 ./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             if (_isElement(sequence))
35             {
36                 return array;
37             }
38             bool hasElements;
39             do
40             {
41                 length *= 2;
42 #if USEARRAYPOOL
43                 var nextArray = ArrayPool.Allocate<ulong>(length);
44 #else
45                 var nextArray = new TLink[length];
46 #endif
47                 hasElements = false;
48                 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 }

```

1.88 ./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 }
```

1.89 ./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         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
24             ↪ stack, links.IsPartialPoint)
25         {
26         }
27
28         public IEnumerable<TLink> Walk(TLink sequence)
29         {
30             _stack.Clear();
31             var element = sequence;
32             if (IsElement(element))
33             {
34                 yield return element;
35             }
36             else
37             {
38             }
```

```

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 }

```

1.90 ./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 =
11             ↳ EqualityComparer<TLink>.Default;
12
13         private readonly ILinks<TLink> _links;
14         private readonly TLink _stack;
15
16         public bool IsEmpty => _equalityComparer.Equals(Peek(), _stack);
17
18         public Stack(ILinks<TLink> links, TLink stack)
19         {
20             _links = links;
21             _stack = stack;
22         }
23
24         private TLink GetStackMarker() => _links.GetSource(_stack);
25
26         private TLink GetTop() => _links.GetTarget(_stack);
27
28         public TLink Peek() => _links.GetTarget(GetTop());
29
30         public TLink Pop()
31         {
32             var element = Peek();
33             if (!_equalityComparer.Equals(element, _stack))
34             {
35                 var top = GetTop();
36                 var previousTop = _links.GetSource(top);
37                 _links.Update(_stack, GetStackMarker(), previousTop);
38                 _links.Delete(top);
39             }
40             return element;
41         }
42     }
43 }

```

```

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

```

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

```

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

```

1.92 ./Platform.Data.Doublents/SynchronizedLinks.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Data.Doublents;
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.Doublents
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         public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
36             ↳ IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
37             ↳ restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
38         public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
39             ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
40         public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
41             ↳ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
42             ↳ Unsync.Update);
43         public void Delete(IList<TLinkAddress> restrictions) =>
44             ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
45
46         //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
47         ↳ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
48         //{
49         //    if (restriction != null && substitution != null &&
50         ↳ !substitution.EqualTo(restriction))
51         //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
52         ↳ substitution, substitutedHandler, Unsync.Trigger);
53         //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
54         ↳ substitutedHandler, Unsync.Trigger);
55     }
56 }

```

```

44     //}
45 }
46 }

```

1.93 ./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.Doublets.Unicode;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets
10 {
11     public static class UInt64LinksExtensions
12     {
13         public static readonly LinksConstants<ulong> Constants =
14             ↳ Default<LinksConstants<ulong>>.Instance;
15
16         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
17
18         public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
19         {
20             if (sequence == null)
21             {
22                 return false;
23             }
24             var constants = links.Constants;
25             for (var i = 0; i < sequence.Length; i++)
26             {
27                 if (sequence[i] == constants.Any)
28                 {
29                     return true;
30                 }
31             }
32             return false;
33         }
34
35         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
36             ↳ Func<Link<ulong>, bool> isElement, bool renderIndex = false, bool renderDebug =
37             ↳ false)
38         {
39             var sb = new StringBuilder();
40             var visited = new HashSet<ulong>();
41             links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
42                 ↳ innerSb.Append(link.Index), renderIndex, renderDebug);
43             return sb.ToString();
44         }
45
46         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
47             ↳ Func<Link<ulong>, bool> isElement, Action<StringBuilder, Link<ulong>> appendElement,
48             ↳ bool renderIndex = false, bool renderDebug = false)
49         {
50             var sb = new StringBuilder();
51             var visited = new HashSet<ulong>();
52             links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
53                 ↳ renderDebug);
54             return sb.ToString();
55         }
56
57         public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
58             ↳ HashSet<ulong> visited, ulong linkIndex, Func<Link<ulong>, bool> isElement,
59             ↳ Action<StringBuilder, Link<ulong>> appendElement, bool renderIndex = false, bool
60             ↳ renderDebug = false)
61         {
62             if (sb == null)
63             {
64                 throw new ArgumentNullException(nameof(sb));
65             }
66             if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
67                 ↳ Constants.Itself)
68             {
69                 return;
70             }
71             if (links.Exists(linkIndex))
72             {
73                 if (visited.Add(linkIndex))
74                 {
75

```



```

64         sb.Append('(');
65         var link = new Link<ulong>(links.GetLink(linkIndex));
66         if (renderIndex)
67         {
68             sb.Append(link.Index);
69             sb.Append(':');
70         }
71         if (link.Source == link.Index)
72         {
73             sb.Append(link.Index);
74         }
75         else
76         {
77             var source = new Link<ulong>(links.GetLink(link.Source));
78             if (isElement(source))
79             {
80                 appendElement(sb, source);
81             }
82             else
83             {
84                 links.AppendStructure(sb, visited, source.Index, isElement,
85                     ↪ appendElement, renderIndex);
86             }
87         }
88         sb.Append(' ');
89         if (link.Target == link.Index)
90         {
91             sb.Append(link.Index);
92         }
93         else
94         {
95             var target = new Link<ulong>(links.GetLink(link.Target));
96             if (isElement(target))
97             {
98                 appendElement(sb, target);
99             }
100             else
101             {
102                 links.AppendStructure(sb, visited, target.Index, isElement,
103                     ↪ appendElement, renderIndex);
104             }
105         }
106         sb.Append(')');
107     }
108     else
109     {
110         if (renderDebug)
111         {
112             sb.Append('*');
113         }
114         sb.Append(linkIndex);
115     }
116 }
117 else
118 {
119     if (renderDebug)
120     {
121         sb.Append('~');
122     }
123     sb.Append(linkIndex);
124 }
125 }

```

1.94 ./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using System.IO;
5  using System.Runtime.CompilerServices;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Platform.Disposables;
9  using Platform.Timestamps;
10 using Platform.Unsafe;
11 using Platform.IO;
12 using Platform.Data.Doublets.Decorators;
13 using Platform.Exceptions;

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

385         {
386             ex.Ignore();
387         }
388     }
389
390     #region DisposalBase
391
392     protected override void Dispose(bool manual, bool wasDisposed)
393     {
394         if (!wasDisposed)
395         {
396             DisposeTransitions();
397         }
398         base.Dispose(manual, wasDisposed);
399     }
400
401     #endregion
402 }
403 }

```

1.95 ./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             public TLink Convert(char source)
21             {
22                 var unaryNumber = _addressToNumberConverter.Convert((Integer<TLink>)source);
23                 return Links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
24             }
25     }
26 }

```

1.96 ./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             public TLink Convert(string source)
27             {
28                 var elements = new TLink[source.Length];
29                 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 }

```

1.97 ./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;
66
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         public static char FromLinkToChar(ulong link) => (char)(link - 1);
69

```



```

68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 public static bool IsCharLink(ulong link) => link <= MapSize;
70
71 public static string FromLinksToString(IList<ulong> linksList)
72 {
73     var sb = new StringBuilder();
74     for (int i = 0; i < linksList.Count; i++)
75     {
76         sb.Append(FromLinkToChar(linksList[i]));
77     }
78     return sb.ToString();
79 }
80
81 public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
82 {
83     var sb = new StringBuilder();
84     if (links.Exists(link))
85     {
86         StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
87             x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
88             ↪ element =>
89             {
90                 sb.Append(FromLinkToChar(element));
91                 return true;
92             });
93     }
94     return sb.ToString();
95 }
96
97 public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
98     ↪ chars.Length);
99
100 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
101 {
102     // char array to ulong array
103     var linksSequence = new ulong[count];
104     for (var i = 0; i < count; i++)
105     {
106         linksSequence[i] = FromCharToLink(chars[i]);
107     }
108     return linksSequence;
109 }
110
111 public static ulong[] FromStringToLinkArray(string sequence)
112 {
113     // char array to ulong array
114     var linksSequence = new ulong[sequence.Length];
115     for (var i = 0; i < sequence.Length; i++)
116     {
117         linksSequence[i] = FromCharToLink(sequence[i]);
118     }
119     return linksSequence;
120 }
121
122 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
123 {
124     var result = new List<ulong[]>();
125     var offset = 0;
126     while (offset < sequence.Length)
127     {
128         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
129         var relativeLength = 1;
130         var absoluteLength = offset + relativeLength;
131         while (absoluteLength < sequence.Length &&
132             currentCategory ==
133             ↪ CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
134         {
135             relativeLength++;
136             absoluteLength++;
137         }
138         // char array to ulong array
139         var innerSequence = new ulong[relativeLength];
140         var maxLength = offset + relativeLength;
141         for (var i = offset; i < maxLength; i++)
142         {
143             innerSequence[i - offset] = FromCharToLink(sequence[i]);
144         }
145         result.Add(innerSequence);
146         offset += relativeLength;
147     }
148 }

```

```

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 }

```

1.98 ./Platform.Data.Doublents/Unicode/UnicodeSequenceCriterionMatcher.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.Doublents.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 }

```

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

```

1 using System;
2 using System.Linq;
3 using Platform.Interfaces;
4 using Platform.Converters;
5 using Platform.Data.Doublents.Sequences.Walkers;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublents.Unicode

```

```

10 {
11     public class UnicodeSequenceToStringConverter<TLink> : LinksOperatorBase<TLink>,
        ↳ IConverter<TLink, string>
12     {
13         private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
14         private readonly ISequenceWalker<TLink> _sequenceWalker;
15         private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;
16
17         public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
            ↳ unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
            ↳ IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
18         {
19             _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
20             _sequenceWalker = sequenceWalker;
21             _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
22         }
23
24         public string Convert(TLink source)
25         {
26             if(!_unicodeSequenceCriterionMatcher.IsMatched(source))
27             {
28                 throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                    ↳ not a unicode sequence.");
29             }
30             var sequence = Links.GetSource(source);
31             var charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.
                ↳ Convert).ToArray();
32             return new string(charArray);
33         }
34     }
35 }

```

1.100 ./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>,
        ↳ ICriterionMatcher<TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
11         private readonly TLink _unicodeSymbolMarker;
12         public UnicodeSymbolCriterionMatcher(ILinks<TLink> links, TLink unicodeSymbolMarker) :
            ↳ base(links) => _unicodeSymbolMarker = unicodeSymbolMarker;
13         public bool IsMatched(TLink link) => _equalityComparer.Equals(Links.GetTarget(link),
            ↳ _unicodeSymbolMarker);
14     }
15 }

```

1.101 ./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>,
        ↳ IConverter<TLink, char>
11     {
12         private readonly IConverter<TLink> _numberToAddressConverter;
13         private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;
14
15         public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
            ↳ numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
            ↳ base(links)
16         {
17             _numberToAddressConverter = numberToAddressConverter;
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(source));
↳ ce(source));
28     }
29 }
30 }

```

1.102 ./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
65             Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {result}");
66         }
67     }
68 }

```

1.103 ./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 }

```

1.104 ./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().Te
37                 ↪ stMultipleRandomCreationsAndDeletions(100));
38             Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
39                 ↪ MultipleRandomCreationsAndDeletions(100));
40             Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
41                 ↪ tMultipleRandomCreationsAndDeletions(100));
42         }
43
44         private static void Using<TLink>(Action<ILinks<TLink>> action)
45         {
46             using (var scope = new Scope<Types<HeapResizableDirectMemory,
47                 ↪ ResizableDirectMemoryLinks<TLink>>>())
48             {
49                 action(scope.Use<ILinks<TLink>>());
50             }
51         }
52     }
53 }

```

```
46     }
47 }
```

1.105 ./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 }
```

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

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

```

54 Volutpat odio facilisis mauris sit amet.
55 Turpis egestas pretium aenean pharetra magna ac placerat.
56 Fermentum dui faucibus in ornare quam viverra orci sagittis eu.
57 Porttitor leo a diam sollicitudin tempor id eu.
58 Volutpat sed cras ornare arcu dui.
59 Ut aliquam purus sit amet luctus venenatis lectus magna.
60 Aliquet risus feugiat in ante metus dictum at.
61 Mattis nunc sed blandit libero.
62 Elit pellentesque habitant morbi tristique senectus et netus.
63 Nibh sit amet commodo nulla facilisi nullam vehicula ipsum a.
64 Enim sit amet venenatis urna cursus eget nunc scelerisque viverra.
65 Amet venenatis urna cursus eget nunc scelerisque viverra mauris in.
66 Diam donec adipiscing tristique risus nec feugiat.
67 Pulvinar mattis nunc sed blandit libero volutpat.
68 Cras fermentum odio eu feugiat pretium nibh ipsum.
69 In nulla posuere sollicitudin aliquam ultrices sagittis orci a.
70 Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.
71 A iaculis at erat pellentesque.
72 Morbi blandit cursus risus at ultrices mi tempus imperdiet nulla.
73 Eget lorem dolor sed viverra ipsum nunc.
74 Leo a diam sollicitudin tempor id eu.
75 Interdum consectetur libero id faucibus nisl tincidunt eget nullam non.";
76
77 [Fact]
78 public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
79 {
80     using (var scope = new TempLinksTestScope(useSequences: false))
81     {
82         var links = scope.Links;
83         var constants = links.Constants;
84
85         links.UseUnicode();
86
87         var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
88
89         var meaningRoot = links.CreatePoint();
90         var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
91         var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
92         var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
93             ↳ constants.Itself);
94
95         var unaryNumberToAddressConverter = new
96             ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
97         var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
98         var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
99             ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
100         var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
101             ↳ frequencyPropertyMarker, frequencyMarker);
102         var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
103             ↳ frequencyPropertyOperator, frequencyIncrementer);
104         var linkToItsFrequencyNumberConverter = new
105             ↳ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
106             ↳ unaryNumberToAddressConverter);
107         var sequenceToItsLocalElementLevelsConverter = new
108             ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
109             ↳ linkToItsFrequencyNumberConverter);
110         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
111             ↳ sequenceToItsLocalElementLevelsConverter);
112
113         var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
114             ↳ Walker = new LeveledSequenceWalker<ulong>(links) });
115
116         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
117             ↳ index, optimalVariantConverter);
118     }
119 }
120
121 [Fact]
122 public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
123 {
124     using (var scope = new TempLinksTestScope(useSequences: false))
125     {
126         var links = scope.Links;
127
128         links.UseUnicode();
129
130         var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
131
132         var totalSequenceSymbolFrequencyCounter = new
133             ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links);

```



```

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);
190
191         var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
192             ↳ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
193
194         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].ShiftRight());
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 garbadage
206     unicodeSequences.CompactAll();
207
208     var linksCountAfterCompactification = links.Count();
209
210     Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
211 }

```

1.107 ./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
34                 var sw1 = Stopwatch.StartNew();
35                 var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
36
37                 var sw2 = Stopwatch.StartNew();
38                 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 }

```

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

```

1  using System.IO;
2  using Xunit;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using Platform.Data.Doublets.ResizableDirectMemory.Specific;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public static class ResizableDirectMemoryLinksTests
10     {
11         private static readonly LinksConstants<ulong> _constants =
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                 {
44                     using (var memory = new
45                         ↳ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
46                     using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
47                         ↳ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
48                     {

```

```

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

```

1.109 ./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>>();
53                 Assert.IsType<ResizableDirectMemoryLinks<ulong>>(links);
54             }
55         }
56     }
57 }

```

1.110 ./Platform.Data.Doublets.Tests/SequencesTests.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Linq;
5  using Xunit;
6  using Platform.Collections;
7  using Platform.Collections.Arrays;
8  using Platform.Random;
9  using Platform.IO;
10 using Platform.Singletons;
11 using Platform.Data.Doublets.Sequences;
12 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
13 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
14 using Platform.Data.Doublets.Sequences.Converters;
15 using Platform.Data.Doublets.Unicode;
16
17 namespace Platform.Data.Doublets.Tests
18 {
19     public static class SequencesTests
20     {
21         private static readonly LinksConstants<ulong> _constants =
22             ↳ Default<LinksConstants<ulong>>.Instance;
23
24         static SequencesTests()
25         {
26             // Trigger static constructor to not mess with performance measurements
27             _ = BitString.GetBitMaskFromIndex(1);
28         }
29
30         [Fact]
31         public static void CreateAllVariantsTest()
32         {
33             const long sequenceLength = 8;
34
35             using (var scope = new TempLinksTestScope(useSequences: true))
36             {
37                 var links = scope.Links;
38                 var sequences = scope.Sequences;
39
40                 var sequence = new ulong[sequenceLength];
41                 for (var i = 0; i < sequenceLength; i++)
42                 {
43                     sequence[i] = links.Create();
44                 }
45
46                 var sw1 = Stopwatch.StartNew();
47                 var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
48
49                 var sw2 = Stopwatch.StartNew();
50                 var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
51
52                 Assert.True(results1.Count > results2.Length);
53                 Assert.True(sw1.Elapsed > sw2.Elapsed);
54
55                 for (var i = 0; i < sequenceLength; i++)
56                 {
57                     links.Delete(sequence[i]);
58                 }
59
60                 Assert.True(links.Count() == 0);
61             }
62
63             //[Fact]
64             //public void CUDTest()
65             //{
66             //    var tempFilename = Path.GetTempFileName();
67
68             //    const long sequenceLength = 8;
69
70             //    const ulong itself = LinksConstants.Itself;
71
72             //    using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
73             //        ↳ DefaultLinksSizeStep))
74             //    using (var links = new Links(memoryAdapter))
75             //    {
76             //        var sequence = new ulong[sequenceLength];
77             //        for (var i = 0; i < sequenceLength; i++)
78             //            sequence[i] = links.Create(itself, itself);
79

```

```

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

```

```

159 public static void BalancedVariantSearchTest()
160 {
161     const long sequenceLength = 200;
162
163     using (var scope = new TempLinksTestScope(useSequences: true))
164     {
165         var links = scope.Links;
166         var sequences = scope.Sequences;
167
168         var sequence = new ulong[sequenceLength];
169         for (var i = 0; i < sequenceLength; i++)
170         {
171             sequence[i] = links.Create();
172         }
173
174         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
175
176         var sw1 = Stopwatch.StartNew();
177         var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
178
179         var sw2 = Stopwatch.StartNew();
180         var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
181
182         var sw3 = Stopwatch.StartNew();
183         var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
184
185         // На количестве в 200 элементов это будет занимать вечность
186         //var sw4 = Stopwatch.StartNew();
187         //var searchResults4 = sequences.Each(sequence); sw4.Stop();
188
189         Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
190
191         Assert.True(searchResults3.Count == 1 && balancedVariant ==
192             ↪ searchResults3.First());
193
194         //Assert.True(sw1.Elapsed < sw2.Elapsed);
195
196         for (var i = 0; i < sequenceLength; i++)
197         {
198             links.Delete(sequence[i]);
199         }
200     }
201
202     [Fact]
203     public static void AllPartialVariantsSearchTest()
204     {
205         const long sequenceLength = 8;
206
207         using (var scope = new TempLinksTestScope(useSequences: true))
208         {
209             var links = scope.Links;
210             var sequences = scope.Sequences;
211
212             var sequence = new ulong[sequenceLength];
213             for (var i = 0; i < sequenceLength; i++)
214             {
215                 sequence[i] = links.Create();
216             }
217
218             var createResults = sequences.CreateAllVariants2(sequence);
219
220             //var createResultsStrings = createResults.Select(x => x + ": " +
221             ↪ sequences.FormatSequence(x)).ToList();
222             //Global.Trash = createResultsStrings;
223
224             var partialSequence = new ulong[sequenceLength - 2];
225
226             Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
227
228             var sw1 = Stopwatch.StartNew();
229             var searchResults1 =
230             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
231
232             var sw2 = Stopwatch.StartNew();
233             var searchResults2 =
234             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
235
236             //var sw3 = Stopwatch.StartNew();

```

```

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         Assert.True(searchResults2.Count == 1 && balancedVariant ==
294         ↪ searchResults2.First());
295
296         for (var i = 0; i < sequenceLength; i++)
297         {
298             links.Delete(sequence[i]);
299         }
300     }
301 }
302
303 [Fact(Skip = "Correct implementation is pending")]
304 public static void PatternMatchTest()
305 {
306     var zeroOrMany = Sequences.Sequences.ZeroOrMany;
307
308     using (var scope = new TempLinksTestScope(useSequences: true))

```



```

307     {
308         var links = scope.Links;
309         var sequences = scope.Sequences;
310
311         var e1 = links.Create();
312         var e2 = links.Create();
313
314         var sequence = new[]
315         {
316             e1, e2, e1, e2 // mama / papa
317         };
318
319         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
320
321         var balancedVariant = balancedVariantConverter.Convert(sequence);
322
323         // 1: [1]
324         // 2: [2]
325         // 3: [1,2]
326         // 4: [1,2,1,2]
327
328         var doublet = links.GetSource(balancedVariant);
329
330         var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
331
332         Assert.True(matchedSequences1.Count == 0);
333
334         var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
335
336         Assert.True(matchedSequences2.Count == 0);
337
338         var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
339
340         Assert.True(matchedSequences3.Count == 0);
341
342         var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
343
344         Assert.Contains(doublet, matchedSequences4);
345         Assert.Contains(balancedVariant, matchedSequences4);
346
347         for (var i = 0; i < sequence.Length; i++)
348         {
349             links.Delete(sequence[i]);
350         }
351     }
352 }
353
354 [Fact]
355 public static void IndexTest()
356 {
357     using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
358         ↪ true }, useSequences: true))
359     {
360         var links = scope.Links;
361         var sequences = scope.Sequences;
362         var index = sequences.Options.Index;
363
364         var e1 = links.Create();
365         var e2 = links.Create();
366
367         var sequence = new[]
368         {
369             e1, e2, e1, e2 // mama / papa
370         };
371
372         Assert.False(index.MightContain(sequence));
373
374         index.Add(sequence);
375
376         Assert.True(index.MightContain(sequence));
377     }
378 }
379
380 /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%
381 ↪ 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
382 ↪ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
383 private static readonly string _exampleText =
384     @"([english
385     ↪ version] (https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))

```

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

384

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

386

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

388

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

390

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

392

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

394

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

396

397 [![две белые точки, чёрная вертикальная
→ линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png "две
→ белые точки, чёрная вертикальная
→ линия")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)

398

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

400

401 [![белая вертикальная линия, чёрный
→ круг](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png "белая
→ вертикальная линия, чёрный
→ круг")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png)

402

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

404

405 [![белый круг, чёрная горизонтальная
→ линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png "белый
→ круг, чёрная горизонтальная
→ линия")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png)

406

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

408

409 [![белая горизонтальная линия, чёрная горизонтальная
→ стрелка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png
→ "белая горизонтальная линия, чёрная горизонтальная
→ стрелка")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png)

410

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

412

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

414

```

415 Допустим у нас есть смысл ""связать"" и смысл ""направления"", много ли это нам даёт? Много ли
    ↳ вариантов интерпретации? А что если уточнить, каким именно образом выполнена связь? Что если
    ↳ можно задать ей чёткий, конкретный смысл? Что это будет? Тип? Глагол? Связка? Действие?
    ↳ Трансформация? Переход из состояния в состояние? Или всё это и есть объект, суть которого в
    ↳ его конечном состоянии, если конечно конец определён направлением?
416
417 [![белая обычная и направленная связи, чёрная типизированная
    ↳ связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png ""белая
    ↳ обычная и направленная связи, чёрная типизированная
    ↳ связь"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png)
418
419 А что если всё это время, мы смотрели на суть как бы снаружи? Можно ли взглянуть на это изнутри?
    ↳ Что будет внутри объектов? Объекты ли это? Или это связи? Может ли эта структура описать
    ↳ сама себя? Но что тогда получится, разве это не рекурсия? Может это фрактал?
420
421 [![белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная
    ↳ связь с рекурсивной внутренней
    ↳ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png
    ↳ ""белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с рекурсивной внутренней структурой"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png)
422
423 На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
    ↳ рекурсии или фрактала?
424
425 [![белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с двойной рекурсивной внутренней
    ↳ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png
    ↳ ""белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с двойной рекурсивной внутренней структурой"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)
426
427 Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
    ↳ Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
428
429 [![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности,
    ↳ чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://
    ↳ raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png ""белая обычная и
    ↳ направленная связи со структурой из 8 цветных элементов последовательности, чёрная
    ↳ типизированная связь со структурой из 8 цветных элементов последовательности"")](https://raw
    ↳ .githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)
430
431 ...
432
433 [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-anim
    ↳ ation-500.gif
    ↳ ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
    ↳ -animation-500.gif)";
434
435     private static readonly string _exampleLoremIpsumText =
436         @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
            ↳ incididunt ut labore et dolore magna aliqua.
437 Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
    ↳ consequat.";
438
439     [Fact]
440     public static void CompressionTest()
441     {
442         using (var scope = new TempLinksTestScope(useSequences: true))
443         {
444             var links = scope.Links;
445             var sequences = scope.Sequences;
446
447             var e1 = links.Create();
448             var e2 = links.Create();
449
450             var sequence = new[]
451             {
452                 e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
453             };
454
455             var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
456             var totalSequenceSymbolFrequencyCounter = new
                ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
457             var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
                ↳ totalSequenceSymbolFrequencyCounter);
458             var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
                ↳ balancedVariantConverter, doubletFrequenciesCache);
459

```

```

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

```

```

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

```

```

591 {
592     var sequence1 = compressed1[i];
593     var sequence2 = compressed2[i];
594     var sequence3 = compressed3[i];
595
596     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
597         ↪ scope1.Links.Unsync);
598
599     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
600         ↪ scope2.Links.Unsync);
601
602     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
603         ↪ scope3.Links.Unsync);
604
605     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
606         ↪ link.IsPartialPoint());
607     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
608         ↪ link.IsPartialPoint());
609     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
610         ↪ link.IsPartialPoint());
611
612     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
613     ↪ arrays[i].Length > 3)
614     //    Assert.False(structure1 == structure2);
615     //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
616     ↪ arrays[i].Length > 3)
617     //    Assert.False(structure3 == structure2);
618
619     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
620     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
621 }
622
623 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
624     ↪ totalCharacters);
625 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
626     ↪ totalCharacters);
627 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
628     ↪ totalCharacters);
629
630 Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
631     ↪ totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
632     ↪ totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
633     ↪ totalCharacters}");
634
635 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
636     ↪ scope2.Links.Unsync.Count() - initialCount2);
637 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
638     ↪ scope2.Links.Unsync.Count() - initialCount2);
639
640 var duplicateProvider1 = new
641     ↪ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
642 var duplicateProvider2 = new
643     ↪ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
644 var duplicateProvider3 = new
645     ↪ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
646
647 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
648 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
649 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
650
651 var duplicates1 = duplicateCounter1.Count();
652
653 ConsoleHelpers.Debug("-----");
654
655 var duplicates2 = duplicateCounter2.Count();
656
657 ConsoleHelpers.Debug("-----");
658
659 var duplicates3 = duplicateCounter3.Count();
660
661 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
662
663 linkFrequenciesCache1.ValidateFrequencies();
664 linkFrequenciesCache3.ValidateFrequencies();
665
666 }
667
668 }
669

```

```

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

```

```

728         if (first == second)
729         {
730             compressed2[i] = first;
731         }
732     }
733
734     var elapsed2 = sw2.Elapsed;
735
736     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
737     ↪ {elapsed2}");
738
739     Assert.True(elapsed1 > elapsed2);
740
741     // Checks
742     for (int i = START; i < END; i++)
743     {
744         var sequence1 = compressed1[i];
745         var sequence2 = compressed2[i];
746
747         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
748         {
749             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
750             ↪ scope1.Links);
751
752             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
753             ↪ scope2.Links);
754
755             //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
756             ↪ link.IsPartialPoint());
757             //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
758             ↪ link.IsPartialPoint());
759
760             //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
761             ↪ arrays[i].Length > 3)
762             //    Assert.False(structure1 == structure2);
763
764             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
765         }
766     }
767
768     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
769     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
770
771     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
772     ↪ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
773     ↪ totalCharacters}}");
774
775     Assert.True(scope1.Links.Count() <= scope2.Links.Count());
776
777     //compressor1.ValidateFrequencies();
778 }
779
780 [Fact]
781 public static void RandomNumbersCompressionQualityTest()
782 {
783     const ulong N = 500;
784
785     //const ulong minNumbers = 10000;
786     //const ulong maxNumbers = 20000;
787
788     //var strings = new List<string>();
789
790     //for (ulong i = 0; i < N; i++)
791     //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
792     ↪ maxNumbers).ToString());
793
794     var strings = new List<string>();
795
796     for (ulong i = 0; i < N; i++)
797     {
798         strings.Add(RandomHelpers.Default.NextUInt64().ToString());
799     }
800
801     strings = strings.Distinct().ToList();
802
803     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
804     var totalCharacters = arrays.Select(x => x.Length).Sum();

```



```

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

```

```

870 //const long sequenceLength = 5; //100% bug
871 const long sequenceLength = 4; //100% bug
872 //const long sequenceLength = 3; //100% _no_bug (ok)
873
874 using (var scope = new TempLinksTestScope(useSequences: true))
875 {
876     var links = scope.Links;
877     var sequences = scope.Sequences;
878
879     var sequence = new ulong[sequenceLength];
880     for (var i = 0; i < sequenceLength; i++)
881     {
882         sequence[i] = links.Create();
883     }
884
885     var createResults = sequences.CreateAllVariants2(sequence);
886
887     Global.Trash = createResults;
888
889     for (var i = 0; i < sequenceLength; i++)
890     {
891         links.Delete(sequence[i]);
892     }
893 }
894
895 [Fact]
896 public static void AllPossibleConnectionsTest()
897 {
898     const long sequenceLength = 5;
899
900     using (var scope = new TempLinksTestScope(useSequences: true))
901     {
902         var links = scope.Links;
903         var sequences = scope.Sequences;
904
905         var sequence = new ulong[sequenceLength];
906         for (var i = 0; i < sequenceLength; i++)
907         {
908             sequence[i] = links.Create();
909         }
910
911         var createResults = sequences.CreateAllVariants2(sequence);
912         var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
913
914         for (var i = 0; i < 1; i++)
915         {
916             var sw1 = Stopwatch.StartNew();
917             var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
918
919             var sw2 = Stopwatch.StartNew();
920             var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
921
922             var sw3 = Stopwatch.StartNew();
923             var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
924
925             var sw4 = Stopwatch.StartNew();
926             var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
927
928             Global.Trash = searchResults3;
929             Global.Trash = searchResults4; //-V3008
930
931             var intersection1 = createResults.Intersect(searchResults1).ToList();
932             Assert.True(intersection1.Count == createResults.Length);
933
934             var intersection2 = reverseResults.Intersect(searchResults1).ToList();
935             Assert.True(intersection2.Count == reverseResults.Length);
936
937             var intersection0 = searchResults1.Intersect(searchResults2).ToList();
938             Assert.True(intersection0.Count == searchResults2.Count);
939
940             var intersection3 = searchResults2.Intersect(searchResults3).ToList();
941             Assert.True(intersection3.Count == searchResults3.Count);
942
943             var intersection4 = searchResults3.Intersect(searchResults4).ToList();
944             Assert.True(intersection4.Count == searchResults4.Count);
945         }
946
947         for (var i = 0; i < sequenceLength; i++)
948         {
949

```

```

950         links.Delete(sequence[i]);
951     }
952 }
953 }
954
955 [Fact(Skip = "Correct implementation is pending")]
956 public static void CalculateAllUsagesTest()
957 {
958     const long sequenceLength = 3;
959
960     using (var scope = new TempLinksTestScope(useSequences: true))
961     {
962         var links = scope.Links;
963         var sequences = scope.Sequences;
964
965         var sequence = new ulong[sequenceLength];
966         for (var i = 0; i < sequenceLength; i++)
967         {
968             sequence[i] = links.Create();
969         }
970
971         var createResults = sequences.CreateAllVariants2(sequence);
972
973         //var reverseResults =
974         ↪ sequences.CreateAllVariants2(sequence.Reverse().ToArray());
975
976         for (var i = 0; i < 1; i++)
977         {
978             var linksTotalUsages1 = new ulong[links.Count() + 1];
979
980             sequences.CalculateAllUsages(linksTotalUsages1);
981
982             var linksTotalUsages2 = new ulong[links.Count() + 1];
983
984             sequences.CalculateAllUsages2(linksTotalUsages2);
985
986             var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
987             Assert.True(intersection1.Count == linksTotalUsages2.Length);
988
989             for (var i = 0; i < sequenceLength; i++)
990             {
991                 links.Delete(sequence[i]);
992             }
993         }
994     }
995 }
996 }

```

1.111 ./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)
24         {
25             _deleteFiles = deleteFiles;
26             TempFilename = Path.GetTempFileName();
27             TempTransactionLogFilename = Path.GetTempFileName();
28             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 }

```

1.112 ./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 // Update link to reference null (prepare for delete)
51 var updated = links.Update(linkAddress, constants.Null, constants.Null);
52
53 Assert.True(equalityComparer.Equals(updated, linkAddress));
54
55 link = new Link<T>(links.GetLink(linkAddress));
56
57 Assert.True(equalityComparer.Equals(link.Source, constants.Null));
58 Assert.True(equalityComparer.Equals(link.Target, constants.Null));
59
60 // Delete link
61 links.Delete(linkAddress);
62
63 Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
64
65 setter = new Setter<T>(constants.Null);
66 links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
67
68 Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
69 }
70
71 public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
72 {
73     // Constants
74     var constants = links.Constants;
75     var equalityComparer = EqualityComparer<T>.Default;
76
77     var h106E = new Hybrid<T>(106L, isExternal: true);
78     var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
79     var h108E = new Hybrid<T>(-108L);
80
81     Assert.Equal(106L, h106E.AbsoluteValue);
82     Assert.Equal(107L, h107E.AbsoluteValue);
83     Assert.Equal(108L, h108E.AbsoluteValue);
84
85     // Create Link (External -> External)
86     var linkAddress1 = links.Create();
87
88     links.Update(linkAddress1, h106E, h108E);
89
90     var link1 = new Link<T>(links.GetLink(linkAddress1));
91
92     Assert.True(equalityComparer.Equals(link1.Source, h106E));
93     Assert.True(equalityComparer.Equals(link1.Target, h108E));
94
95     // Create Link (Internal -> External)
96     var linkAddress2 = links.Create();
97
98     links.Update(linkAddress2, linkAddress1, h108E);
99
100     var link2 = new Link<T>(links.GetLink(linkAddress2));
101
102     Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
103     Assert.True(equalityComparer.Equals(link2.Target, h108E));
104
105     // Create Link (Internal -> Internal)
106     var linkAddress3 = links.Create();
107
108     links.Update(linkAddress3, linkAddress1, linkAddress2);
109
110     var link3 = new Link<T>(links.GetLink(linkAddress3));
111
112     Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
113     Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
114
115     // Search for created link
116     var setter1 = new Setter<T>(constants.Null);
117     links.Each(h106E, h108E, setter1.SetAndReturnFalse);
118
119     Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
120
121     // Search for nonexistent link
122     var setter2 = new Setter<T>(constants.Null);
123     links.Each(h106E, h107E, setter2.SetAndReturnFalse);
124
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.GetOrCreate(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             Assert.True(created == (Integer<TLink>)links.Count());
179             for (var i = 0; i < N; i++)
180             {
181                 TLink link = (Integer<TLink>)(i + 1);
182                 if (links.Exists(link))
183                 {
184                     links.Delete(link);
185                     deleted++;
186                 }
187             }
188             Assert.True((Integer<TLink>)links.Count() == 0);
189         }
190     }
191 }
192 }

```

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

```

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

```

```

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
86                         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope,
87                             ↪ e.TempTransactionLogFilename);
88                     }
89                 }
90             }
91         }
92     }
93 }

```

```

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

```



```

162 [Fact]
163 public static void TransactionUserCodeErrorSomeDataSavedTest()
164 {
165     // User Code Error (Autoreverted), some data saved
166     var itself = _constants.Itself;
167
168     TempLinksTestScope lastScope = null;
169     try
170     {
171         ulong l1;
172         ulong l2;
173
174         using (var scope = new TempLinksTestScope(useLog: true))
175         {
176             var links = scope.Links;
177             l1 = links.CreateAndUpdate(itself, itself);
178             l2 = links.CreateAndUpdate(itself, itself);
179
180             l2 = links.Update(l2, l2, l1, l2);
181
182             links.CreateAndUpdate(l2, itself);
183             links.CreateAndUpdate(l2, itself);
184
185             links.Unsync.DisposeIfPossible();
186
187             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
188                 ↪ scope.TempTransactionLogFilename);
189         }
190
191         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
192             ↪ useLog: true))
193         {
194             var links = scope.Links;
195             var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
196             using (var transaction = transactionsLayer.BeginTransaction())
197             {
198                 l2 = links.Update(l2, l1);
199
200                 links.Delete(l2);
201
202                 ExceptionThrower();
203
204                 transaction.Commit();
205             }
206
207             Global.Trash = links.Count();
208         }
209     }
210     catch
211     {
212         Assert.False(lastScope == null);
213
214         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
215             ↪ Scope.TempTransactionLogFilename);
216
217         lastScope.DeleteFiles();
218     }
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))
234     {
235         using (var transaction = memoryAdapter.BeginTransaction())
236         {
237             var l1 = links.CreateAndUpdate(itself, itself);
238             var l2 = links.CreateAndUpdate(itself, itself);
239

```

```

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                 links.Delete(l2);
343                 ExceptionThrower();
344                 transaction.Commit();
345             }
346
347             Global.Trash = links.Count();
348         }
349     }
350     catch
351     {
352         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
353             ↪ TransactionLogFilename);
354     }
355
356     File.Delete(tempDatabaseFilename);
357     File.Delete(tempTransactionLogFilename);
358 }
359
360 private static void ExceptionThrower() => throw new InvalidOperationException();
361
362 [Fact]
363 public static void PathsTest()
364 {
365     var source = _constants.SourcePart;
366     var target = _constants.TargetPart;
367
368     using (var scope = new TempLinksTestScope())
369     {
370         var links = scope.Links;
371         var l1 = links.CreatePoint();
372         var l2 = links.CreatePoint();
373
374         var r1 = links.GetByKeys(l1, source, target, source);
375         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
376     }
377 }
378
379 [Fact]
380 public static void RecursiveStringFormattingTest()
381 {

```

```

381 using (var scope = new TempLinksTestScope(useSequences: true))
382 {
383     var links = scope.Links;
384     var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
385
386     var a = links.CreatePoint();
387     var b = links.CreatePoint();
388     var c = links.CreatePoint();
389
390     var ab = links.GetOrCreate(a, b);
391     var cb = links.GetOrCreate(c, b);
392     var ac = links.GetOrCreate(a, c);
393
394     a = links.Update(a, c, b);
395     b = links.Update(b, a, c);
396     c = links.Update(c, a, b);
397
398     Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
399     Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
400     Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
401
402     Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
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 private static void DefaultFormatter(StringBuilder sb, ulong link)
421 {
422     sb.Append(link.ToString());
423 }
424
425 #endregion
426
427 #region Performance
428
429 /*
430 public static void RunAllPerformanceTests()
431 {
432     try
433     {
434         links.TestLinksInSteps();
435     }
436     catch (Exception ex)
437     {
438         ex.WriteToConsole();
439     }
440
441     return;
442
443     try
444     {
445         //ThreadPool.SetMaxThreads(2, 2);
446
447         // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
448         ↪ результат
449         // Также это дополнительно помогает в отладке
450         // Увеличивает вероятность попадания информации в кэши
451         for (var i = 0; i < 10; i++)
452         {
453             //0 - 10 ГБ
454             //Каждые 100 МБ срез цифр
455
456             //links.TestGetSourceFunction();
457             //links.TestGetSourceFunctionInParallel();
458             //links.TestGetTargetFunction();
459         }
460     }
461 }

```

```

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
596             long counter = 0;
597
598             //var firstLink = links.First();
599             var firstLink = links.Create();
600
601             var sw = Stopwatch.StartNew();

```

```

602 // Тестируем саму функцию
603 Parallel.For(0, Iterations, x =>
604 {
605     Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
606     //Interlocked.Increment(ref counter);
607 });
608
609 var elapsedTime = sw.Elapsed;
610
611 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
612
613 links.Delete(firstLink);
614
615 ConsoleHelpers.Debug(
616     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
617     ↳ second), counter result: {3}",
618     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
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
650             ↳ second), counter result: {3}",
651             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
652     }
653
654 [Fact(Skip = "performance test")]
655 public static void TestGetTargetInParallel()
656 {
657     using (var scope = new TempLinksTestScope())
658     {
659         var links = scope.Links;
660         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
661             ↳ parallel.", Iterations);
662
663         long counter = 0;
664
665         //var firstLink = links.First();
666         var firstLink = links.Create();
667
668         var sw = Stopwatch.StartNew();
669
670         Parallel.For(0, Iterations, x =>
671         {
672             Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
673             //Interlocked.Increment(ref counter);
674         });
675
676         var elapsedTime = sw.Elapsed;
677
678         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

```

```

678         links.Delete(firstLink);
679
680     ConsoleHelpers.Debug(
681         "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
682         ↳ second), counter result: {3}",
683         iterations, elapsedTime, (long)iterationsPerSecond, counter);
684     }
685 }
686
687 // TODO: Заполнить базу данных перед тестом
688 /*
689 [Fact]
690 public void TestRandomSearchFixed()
691 {
692     var tempFilename = Path.GetTempFileName();
693
694     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
695     ↳ DefaultLinksSizeStep))
696     {
697         long iterations = 64 * 1024 * 1024 /
698     ↳ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
699
700         ulong counter = 0;
701         var maxLink = links.Total;
702
703         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
704
705         var sw = Stopwatch.StartNew();
706
707         for (var i = iterations; i > 0; i--)
708         {
709             var source =
710     ↳ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
711             var target =
712     ↳ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
713
714             counter += links.Search(source, target);
715         }
716
717         var elapsedTime = sw.Elapsed;
718
719         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
720
721         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
722     ↳ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
723     ↳ counter);
724     }
725
726     File.Delete(tempFilename);
727 }*/
728
729 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
730 public static void TestRandomSearchAll()
731 {
732     using (var scope = new TempLinksTestScope())
733     {
734         var links = scope.Links;
735         ulong counter = 0;
736
737         var maxLink = links.Count();
738
739         var iterations = links.Count();
740
741         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
742     ↳ links.Count());
743
744         var sw = Stopwatch.StartNew();
745
746         for (var i = iterations; i > 0; i--)
747         {
748             var linksAddressRange = new
749     ↳ Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);
750
751             var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
752             var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
753
754             counter += links.SearchOrDefault(source, target);
755         }
756     }
757 }

```



```

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
758 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
759 public static void TestEach()
760 {
761     using (var scope = new TempLinksTestScope())
762     {
763         var links = scope.Links;
764
765         var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
766
767         ConsoleHelpers.Debug("Testing Each function.");
768
769         var sw = Stopwatch.StartNew();
770
771         links.Each(counter.IncrementAndReturnTrue);
772
773         var elapsedTime = sw.Elapsed;
774
775         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
776
777         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
778         ↪     links per second)",
779             counter, elapsedTime, (long)linksPerSecond);
780     }
781 }
782
783 /*
784 [Fact]
785 public static void TestForeach()
786 {
787     var tempFilename = Path.GetTempFileName();
788
789     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
790     ↪     DefaultLinksSizeStep))
791     {
792         ulong counter = 0;
793
794         ConsoleHelpers.Debug("Testing foreach through links.");
795
796         var sw = Stopwatch.StartNew();
797
798         //foreach (var link in links)
799         //{
800             //    counter++;
801         //}
802
803         var elapsedTime = sw.Elapsed;
804
805         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
806
807         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
808     ↪     links per second)", counter, elapsedTime, (long)linksPerSecond);
809     }
810
811     File.Delete(tempFilename);
812 }
813 */
814
815 /*
816 [Fact]
817 public static void TestParallelForeach()
818 {
819     var tempFilename = Path.GetTempFileName();
820
821     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
822     ↪     DefaultLinksSizeStep))
823     {
824         long counter = 0;

```

```

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

```

```

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

```

1.114 ./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 }

```

1.115 ./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;
78
79                 var itself = links.Constants.Itself;
80
81                 var meaningRoot = links.CreatePoint();
82                 var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
83                 var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
84                 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    }

```

Index

./Platform.Data.Doublets.Tests/ComparisonTests.cs, 140
./Platform.Data.Doublets.Tests/EqualityTests.cs, 141
./Platform.Data.Doublets.Tests/GenericLinksTests.cs, 142
./Platform.Data.Doublets.Tests/LinksConstantsTests.cs, 143
./Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs, 143
./Platform.Data.Doublets.Tests/ReadSequenceTests.cs, 146
./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs, 147
./Platform.Data.Doublets.Tests/ScopeTests.cs, 148
./Platform.Data.Doublets.Tests/SequencesTests.cs, 149
./Platform.Data.Doublets.Tests/TempLinksTestScope.cs, 163
./Platform.Data.Doublets.Tests/TestExtensions.cs, 164
./Platform.Data.Doublets.Tests/UInt64LinksTests.cs, 166
./Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs, 179
./Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs, 179
./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/ILinks.cs, 13
./Platform.Data.Doublets/ILinksExtensions.cs, 13
./Platform.Data.Doublets/ISynchronizedLinks.cs, 24
./Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs, 24
./Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs, 24
./Platform.Data.Doublets/Link.cs, 25
./Platform.Data.Doublets/LinkExtensions.cs, 28
./Platform.Data.Doublets/LinksOperatorBase.cs, 28
./Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs, 28
./Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConveter.cs, 29
./Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs, 29
./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs, 30
./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs, 31
./Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 32
./Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 32
./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksAvlBalancedTreeMethodsBase.cs, 33
./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSizeBalancedTreeMethodsBase.cs, 37
./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesAvlBalancedTreeMethods.cs, 40
./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesSizeBalancedTreeMethods.cs, 41
./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsAvlBalancedTreeMethods.cs, 42
./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsSizeBalancedTreeMethods.cs, 44
./Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinks.cs, 44
./Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinksBase.cs, 46
./Platform.Data.Doublets/ResizableDirectMemory/Generic/UnusedLinksListMethods.cs, 53
./Platform.Data.Doublets/ResizableDirectMemory/ILinksListMethods.cs, 53
./Platform.Data.Doublets/ResizableDirectMemory/ILinksTreeMethods.cs, 54
./Platform.Data.Doublets/ResizableDirectMemory/LinksHeader.cs, 54
./Platform.Data.Doublets/ResizableDirectMemory/RawLink.cs, 54
./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs, 55
./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs, 57
./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs, 58
./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.cs, 59
./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs, 60
./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs, 61
./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64ResizableDirectMemoryLinks.cs, 62
./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64UnusedLinksListMethods.cs, 64

./Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 64
./Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 65
./Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 68
./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 68
./Platform.Data.Doublets/Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 70
./Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs, 70
./Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs, 70
./Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 71
./Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 72
./Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 72
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 74
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 76
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToltsFrequencyValueConverter.cs, 76
./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 76
./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 77
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 77
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 78
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 78
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 78
./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 79
./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 80
./Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 80
./Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 80
./Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 81
./Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 82
./Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 82
./Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 83
./Platform.Data.Doublets/Sequences/Indexes/Unindex.cs, 84
./Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 84
./Platform.Data.Doublets/Sequences/Sequences.cs, 110
./Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 120
./Platform.Data.Doublets/Sequences/SequencesOptions.cs, 121
./Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 122
./Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 122
./Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 123
./Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 124
./Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 125
./Platform.Data.Doublets/Stacks/Stack.cs, 126
./Platform.Data.Doublets/Stacks/StackExtensions.cs, 127
./Platform.Data.Doublets/SynchronizedLinks.cs, 127
./Platform.Data.Doublets/UInt64LinksExtensions.cs, 128
./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 129
./Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 135
./Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 135
./Platform.Data.Doublets/Unicode/UnicodeMap.cs, 136
./Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs, 138
./Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 138
./Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs, 139
./Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 139