

LinksPlatform's Platform.Data.Doublets Class Library

./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     }

```

./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(IList<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 }

```

./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 }

```

./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 }

```

./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 }

```

./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     [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 }

```

./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 }

```

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

```

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

```

```

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

```

./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 }

```

./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 }

```

./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 }
```

./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 }
```

./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 }

```

./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.Arrays;
6 using Platform.Collections.Lists;
7 using Platform.Data.Universal;
8
9 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Decorators
12 {
13     /// <remarks>
14     /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
15     /// 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).
16     ///
17     /// 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.)

```

```

18  /// </remarks>
19  internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
20  {
21      private static readonly EqualityComparer<TLink> _equalityComparer =
22          ↳ EqualityComparer<TLink>.Default;
23
24      public UniLinks(ILinks<TLink> links) : base(links) { }
25
26      private struct Transition
27      {
28          public IList<TLink> Before;
29          public IList<TLink> After;
30
31          public Transition(IList<TLink> before, IList<TLink> after)
32          {
33              Before = before;
34              After = after;
35          }
36      }
37
38      //public static readonly TLink NullConstant = Use<LinksConstants<TLink>>.Single.Null;
39      //public static readonly IReadOnlyList<TLink> NullLink = new
40      ↳ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
41      ↳ });
42
43      // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
44      ↳ (Links-Expression)
45      public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
46      ↳ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
47      ↳ substitutedHandler)
48      {
49          ///List<Transition> transitions = null;
50          ///if (!restriction.IsNullOrEmpty())
51          ///{
52          ///    // Есть причина делать проход (чтение)
53          ///    if (matchedHandler != null)
54          ///    {
55          ///        if (!substitution.IsNullOrEmpty())
56          ///        {
57          ///            // restriction => { 0, 0, 0 } | { 0 } // Create
58          ///            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
59          ↳ Create / Update
60          ///            // substitution => { 0, 0, 0 } | { 0 } // Delete
61          ///            transitions = new List<Transition>();
62          ///            if (Equals(substitution[Constants.IndexPart], Constants.Null))
63          ///            {
64          ///                // If index is Null, that means we always ignore every other
65          ↳ value (they are also Null by definition)
66          ///                var matchDecision = matchedHandler(, NullLink);
67          ///                if (Equals(matchDecision, Constants.Break))
68          ///                    return false;
69          ///                if (!Equals(matchDecision, Constants.Skip))
70          ///                    transitions.Add(new Transition(matchedLink, newValue));
71          ///            }
72          ///            else
73          ///            {
74          ///                Func<T, bool> handler;
75          ///                handler = link =>
76          ///                {
77          ///                    var matchedLink = Memory.GetLinkValue(link);
78          ///                    var newValue = Memory.GetLinkValue(link);
79          ///                    newValue[Constants.IndexPart] = Constants.Itself;
80          ///                    newValue[Constants.SourcePart] =
81          ↳ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
82          ↳ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
83          ///                    newValue[Constants.TargetPart] =
84          ↳ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
85          ↳ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
86          ///                    var matchDecision = matchedHandler(matchedLink, newValue);
87          ///                    if (Equals(matchDecision, Constants.Break))
88          ///                        return false;
89          ///                    if (!Equals(matchDecision, Constants.Skip))
90          ///                        transitions.Add(new Transition(matchedLink, newValue));
91          ///                    return true;
92          ///                };
93          ///            }
94          ///            if (!Memory.Each(handler, restriction))
95          ///                return Constants.Break;
96          ///        }
97      }

```



```

84         ////    }
85         ////    else
86         ////    {
87         ////        Func<T, bool> handler = link =>
88         ////        {
89         ////            var matchedLink = Memory.GetLinkValue(link);
90         ////            var matchDecision = matchedHandler(matchedLink, matchedLink);
91         ////            return !Equals(matchDecision, Constants.Break);
92         ////        };
93         ////        if (!Memory.Each(handler, restriction))
94         ////            return Constants.Break;
95         ////    }
96         ////    }
97         ////    else
98         ////    {
99         ////        if (substitution != null)
100        ////        {
101        ////            transitions = new List<IList<T>>();
102        ////            Func<T, bool> handler = link =>
103        ////            {
104        ////                var matchedLink = Memory.GetLinkValue(link);
105        ////                transitions.Add(matchedLink);
106        ////                return true;
107        ////            };
108        ////            if (!Memory.Each(handler, restriction))
109        ////                return Constants.Break;
110        ////        }
111        ////        else
112        ////        {
113        ////            return Constants.Continue;
114        ////        }
115        ////    }
116        ////}
117        ////if (substitution != null)
118        ////{
119        ////    // Есть причина делать замену (запись)
120        ////    if (substitutedHandler != null)
121        ////    {
122        ////    }
123        ////    else
124        ////    {
125        ////    }
126        ////}
127        ////return Constants.Continue;
128
129        //if (restriction.IsNullOrEmpty()) // Create
130        //{
131        //    substitution[Constants.IndexPart] = Memory.AllocateLink();
132        //    Memory.SetLinkValue(substitution);
133        //}
134        //else if (substitution.IsNullOrEmpty()) // Delete
135        //{
136        //    Memory.FreeLink(restriction[Constants.IndexPart]);
137        //}
138        //else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
139        //{
140        //    // No need to collect links to list
141        //    // Skip == Continue
142        //    // No need to check substitutedHandler
143        //    if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
144        //        ↪ Constants.Break), restriction))
145        //        return Constants.Break;
146        //}
147        //else // Update
148        //{
149        //    //List<IList<T>> matchedLinks = null;
150        //    if (matchedHandler != null)
151        //    {
152        //        matchedLinks = new List<IList<T>>();
153        //        Func<T, bool> handler = link =>
154        //        {
155        //            var matchedLink = Memory.GetLinkValue(link);
156        //            var matchDecision = matchedHandler(matchedLink);
157        //            if (Equals(matchDecision, Constants.Break))
158        //                return false;
159        //            if (!Equals(matchDecision, Constants.Skip))
160        //                matchedLinks.Add(matchedLink);
161        //            return true;

```

```

161         //     };
162         //     if (!Memory.Each(handler, restriction))
163         //         return Constants.Break;
164         // }
165         // if (!matchedLinks.IsNullOrEmpty())
166         // {
167         //     var totalMatchedLinks = matchedLinks.Count;
168         //     for (var i = 0; i < totalMatchedLinks; i++)
169         //     {
170         //         var matchedLink = matchedLinks[i];
171         //         if (substitutedHandler != null)
172         //         {
173         //             var newValue = new List<T>(); // TODO: Prepare value to update here
174         //             // TODO: Decide is it actually needed to use Before and After
175         //             substitution handling.
176         //             var substitutedDecision = substitutedHandler(matchedLink,
177         //             ↪ newValue);
178         //             if (Equals(substitutedDecision, Constants.Break))
179         //                 return Constants.Break;
180         //             if (Equals(substitutedDecision, Constants.Continue))
181         //             {
182         //                 // Actual update here
183         //                 Memory.SetLinkValue(newValue);
184         //             }
185         //             if (Equals(substitutedDecision, Constants.Skip))
186         //             {
187         //                 // Cancel the update. TODO: decide use separate Cancel
188         //                 ↪ constant or Skip is enough?
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         if (matchHandler != null)
232         {
233             return substitutionHandler(before, after);
234         }
235         return Constants.Continue;
236     }
237     else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
238     {
239         if (patternOrCondition.Count == 1)
240         {
241             var linkToDelete = patternOrCondition[0];
242             var before = Links.GetLink(linkToDelete);
243             if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
244                 ↪ Constants.Break))
245             {
246                 return Constants.Break;
247             }
248             var after = ArrayPool<TLink>.Empty;
249             Links.Update(linkToDelete, Constants.Null, Constants.Null);
250             Links.Delete(linkToDelete);
251             if (matchHandler != null)
252             {
253                 return substitutionHandler(before, after);
254             }
255             return Constants.Continue;
256         }
257         else
258         {
259             throw new NotSupportedException();
260         }
261     }
262     else // Replace / Update
263     {
264         if (patternOrCondition.Count == 1) //-V3125
265         {
266             var linkToUpdate = patternOrCondition[0];
267             var before = Links.GetLink(linkToUpdate);
268             if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
269                 ↪ Constants.Break))
270             {
271                 return Constants.Break;
272             }
273             var after = (IList<TLink>)substitution.ToArray(); //-V3125
274             if (_equalityComparer.Equals(after[0], default))
275             {
276                 after[0] = linkToUpdate;
277             }
278             if (substitution.Count == 1)
279             {
280                 if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
281                 {
282                     after = Links.GetLink(substitution[0]);
283                     Links.Update(linkToUpdate, Constants.Null, Constants.Null);
284                     Links.Delete(linkToUpdate);
285                 }
286             }
287             else if (substitution.Count == 3)
288             {
289                 //Links.Update(after);
290             }
291             else
292             {
293                 throw new NotSupportedException();
294             }
295             if (matchHandler != null)
296             {
297                 return substitutionHandler(before, after);
298             }
299             return Constants.Continue;
300         }
301         else
302         {
303             throw new NotSupportedException();
304         }
305     }
306 }

```

/// <remarks>

```

307     /// IList[IList[IList[T]]]
308     /// | | |
309     /// | | |-----|
310     /// | | |link|
311     /// | | |-----|
312     /// | | |change|
313     /// |-----|
314     /// |changes
315     /// </remarks>
316     public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink>
        ↳ substitution)
317     {
318         var changes = new List<IList<IList<TLink>>>();
319         Trigger(condition, AlwaysContinue, substitution, (before, after) =>
320         {
321             var change = new[] { before, after };
322             changes.Add(change);
323             return Constants.Continue;
324         });
325         return changes;
326     }
327
328     private TLink AlwaysContinue(IList<TLink> linkToMatch) => Constants.Continue;
329 }
330 }

```

./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 }

```

./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 }

```

./Platform.Data.Doublets/Hybrid.cs

```

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

```

```

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

```

./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 }
```

./Platform.Data.Doublets/ILinksExtensions.cs

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

```

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

```



```

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

```

```

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

```

```

254 [MethodImpl(MethodImplOptions.AggressiveInlining)]
255 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
    ↳ Func<IList<TLink>, TLink> handler)
256 {
257     var constants = links.Constants;
258     return links.Each(handler, constants.Any, source, target);
259 }
260
261 [MethodImpl(MethodImplOptions.AggressiveInlining)]
262 public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ restrictions)
263 {
264     long arraySize = (Integer<TLink>)links.Count(restrictions);
265     var array = new IList<TLink>[arraySize];
266     if (arraySize > 0)
267     {
268         var filler = new ArrayFiller<IList<TLink>, TLink>(array,
            ↳ links.Constants.Continue);
269         links.Each(filler.AddAndReturnConstant, restrictions);
270     }
271     return array;
272 }
273
274 [MethodImpl(MethodImplOptions.AggressiveInlining)]
275 public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ restrictions)
276 {
277     long arraySize = (Integer<TLink>)links.Count(restrictions);
278     var array = new TLink[arraySize];
279     if (arraySize > 0)
280     {
281         var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
282         links.Each(filler.AddFirstAndReturnConstant, restrictions);
283     }
284     return array;
285 }
286
287 /// <summary>
288 /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
    ↳ в хранилище связей.
289 /// </summary>
290 /// <param name="links">Хранилище связей.</param>
291 /// <param name="source">Начало связи.</param>
292 /// <param name="target">Конец связи.</param>
293 /// <returns>Значение, определяющее существует ли связь.</returns>
294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
    ↳ => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
    ↳ default) > 0;
296
297 #region Ensure
298 // TODO: May be move to EnsureExtensions or make it both there and here
299
300 [MethodImpl(MethodImplOptions.AggressiveInlining)]
301 public static void EnsureLinkExists<TLink>(this ILinks<TLink> links, IList<TLink>
    ↳ restrictions)
302 {
303     for (var i = 0; i < restrictions.Count; i++)
304     {
305         if (!links.Exists(restrictions[i]))
306         {
307             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                ↳ $"sequence[{i}]");
308         }
309     }
310 }
311
312 [MethodImpl(MethodImplOptions.AggressiveInlining)]
313 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
    ↳ reference, string argumentName)
314 {
315     if (links.Constants.IsInnerReference(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,
323     ↳ IList<TLink> restrictions, string argumentName)
324 {
325     for (int i = 0; i < restrictions.Count; i++)
326     {
327         links.EnsureInnerReferenceExists(restrictions[i], argumentName);
328     }
329
330 [MethodImpl(MethodImplOptions.AggressiveInlining)]
331 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
332     ↳ restrictions)
333 {
334     var equalityComparer = EqualityComparer<TLink>.Default;
335     var any = links.Constants.Any;
336     for (var i = 0; i < restrictions.Count; i++)
337     {
338         if (!equalityComparer.Equals(restrictions[i], any) &&
339             ↳ !links.Exists(restrictions[i]))
340         {
341             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
342                 ↳ $"{sequence[{i}]}");
343         }
344     }
345
346 [MethodImpl(MethodImplOptions.AggressiveInlining)]
347 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
348     ↳ string argumentName)
349 {
350     var equalityComparer = EqualityComparer<TLink>.Default;
351     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
352     {
353         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
354     }
355
356 [MethodImpl(MethodImplOptions.AggressiveInlining)]
357 public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
358     ↳ link, string argumentName)
359 {
360     var equalityComparer = EqualityComparer<TLink>.Default;
361     if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
362     {
363         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
364     }
365 }
366
367 /// <param name="links">Хранилище связей.</param>
368 [MethodImpl(MethodImplOptions.AggressiveInlining)]
369 public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
370     ↳ TLink target)
371 {
372     if (links.Exists(source, target))
373     {
374         throw new LinkWithSameValueAlreadyExistsException();
375     }
376 }
377
378 /// <param name="links">Хранилище связей.</param>
379 public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
380 {
381     if (links.HasUsages(link))
382     {
383         throw new ArgumentLinkHasDependenciesException<TLink>(link);
384     }
385 }
386
387 /// <param name="links">Хранилище связей.</param>
388 public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
389     ↳ addresses) => links.EnsureCreated(links.Create, addresses);
390
391 /// <param name="links">Хранилище связей.</param>
392 public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
393     ↳ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
394
395 /// <param name="links">Хранилище связей.</param>

```

```

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

```

```

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

```

```

525     var substitutionIndex = substitution[constants.IndexPart];
526     if (equalityComparer.Equals(substitutionIndex, default))
527     {
528         substitutionIndex = restrictionsIndex;
529     }
530     var source = substitution[constants.SourcePart];
531     var target = substitution[constants.TargetPart];
532     source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
533     target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
534     return new Link<TLink>(substitutionIndex, source, target);
535 }
536
537 /// <summary>
538 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
539   ↳ с указанными Source (началом) и Target (концом).
540 /// </summary>
541 /// <param name="links">Хранилище связей.</param>
542 /// <param name="source">Индекс связи, которая является началом на создаваемой
543   ↳ связи.</param>
544 /// <param name="target">Индекс связи, которая является концом для создаваемой
545   ↳ связи.</param>
546 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
547 [MethodImpl(MethodImplOptions.AggressiveInlining)]
548 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
549   ↳ target)
550 {
551     var link = links.SearchOrDefault(source, target);
552     if (EqualityComparer<TLink>.Default.Equals(link, default))
553     {
554         link = links.CreateAndUpdate(source, target);
555     }
556     return link;
557 }
558
559 /// <summary>
560 /// Обновляет связь с указанными началом (Source) и концом (Target)
561   ↳ на связь с указанными началом (NewSource) и концом (NewTarget).
562 /// </summary>
563 /// <param name="links">Хранилище связей.</param>
564 /// <param name="source">Индекс связи, которая является началом обновляемой
565   ↳ связи.</param>
566 /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
567 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
568   ↳ выполняется обновление.</param>
569 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
570   ↳ выполняется обновление.</param>
571 /// <returns>Индекс обновлённой связи.</returns>
572 [MethodImpl(MethodImplOptions.AggressiveInlining)]
573 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
574   ↳ TLink target, TLink newSource, TLink newTarget)
575 {
576     var equalityComparer = EqualityComparer<TLink>.Default;
577     var link = links.SearchOrDefault(source, target);
578     if (equalityComparer.Equals(link, default))
579     {
580         return links.CreateAndUpdate(newSource, newTarget);
581     }
582     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
583   ↳ target))
584     {
585         return link;
586     }
587     return links.Update(link, newSource, newTarget);
588 }
589
590 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
591 /// <param name="links">Хранилище связей.</param>
592 /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
593 /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
594 [MethodImpl(MethodImplOptions.AggressiveInlining)]
595 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
596   ↳ target)
597 {
598     var link = links.SearchOrDefault(source, target);
599     if (!EqualityComparer<TLink>.Default.Equals(link, default))
600     {
601         links.Delete(link);
602     }
603 }

```

```

592         return link;
593     }
594     return default;
595 }
596
597 /// <summary>Удаляет несколько связей.</summary>
598 /// <param name="links">Хранилище связей.</param>
599 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
600 [MethodImpl(MethodImplOptions.AggressiveInlining)]
601 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
602 {
603     for (int i = 0; i < deletedLinks.Count; i++)
604     {
605         links.Delete(deletedLinks[i]);
606     }
607 }
608
609 /// <remarks>Before execution of this method ensure that deleted link is detached (all
610 ↪ values - source and target are reset to null) or it might enter into infinite
611 ↪ recursion.</remarks>
612 public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
613 {
614     var anyConstant = links.Constants.Any;
615     var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
616     links.DeleteByQuery(usagesAsSourceQuery);
617     var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
618     links.DeleteByQuery(usagesAsTargetQuery);
619 }
620
621 public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
622 {
623     var count = (Integer<TLink>)links.Count(query);
624     if (count > 0)
625     {
626         var queryResult = new TLink[count];
627         var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
628             ↪ links.Constants.Continue);
629         links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
630         for (var i = (long)count - 1; i >= 0; i--)
631         {
632             links.Delete(queryResult[i]);
633         }
634     }
635 }
636
637 // TODO: Move to Platform.Data
638 public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
639 {
640     var nullConstant = links.Constants.Null;
641     var equalityComparer = EqualityComparer<TLink>.Default;
642     var link = links.GetLink(linkIndex);
643     for (int i = 1; i < link.Count; i++)
644     {
645         if (!equalityComparer.Equals(link[i], nullConstant))
646         {
647             return false;
648         }
649     }
650     return true;
651 }
652
653 // TODO: Create a universal version of this method in Platform.Data (with using of for
654 ↪ loop)
655 public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
656 {
657     var nullConstant = links.Constants.Null;
658     var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
659     links.Update(updateRequest);
660 }
661
662 // TODO: Create a universal version of this method in Platform.Data (with using of for
663 ↪ loop)
664 public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
665 {
666     if (!links.AreValuesReset(linkIndex))
667     {
668         links.ResetValues(linkIndex);
669     }
670 }

```



```

665 }
666
667 /// <summary>
668 /// Merging two usages graphs, all children of old link moved to be children of new link
669   ↳ or deleted.
670 /// </summary>
671 public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
672   ↳ TLink newLinkIndex)
673 {
674     var equalityComparer = EqualityComparer<TLink>.Default;
675     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
676     {
677         var constants = links.Constants;
678         var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
679           ↳ constants.Any);
680         long usagesAsSourceCount = (Integer<TLink>)links.Count(usagesAsSourceQuery);
681         var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
682           ↳ oldLinkIndex);
683         long usagesAsTargetCount = (Integer<TLink>)links.Count(usagesAsTargetQuery);
684         var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
685           ↳ usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
686         if (!isStandalonePoint)
687         {
688             var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
689             if (totalUsages > 0)
690             {
691                 var usages = ArrayPool.Allocate<TLink>(totalUsages);
692                 var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
693                   ↳ links.Constants.Continue);
694                 var i = 0L;
695                 if (usagesAsSourceCount > 0)
696                 {
697                     links.Each(usagesFiller.AddFirstAndReturnConstant,
698                       ↳ usagesAsSourceQuery);
699                     for (; i < usagesAsSourceCount; i++)
700                     {
701                         var usage = usages[i];
702                         if (!equalityComparer.Equals(usage, oldLinkIndex))
703                         {
704                             links.Update(usage, newLinkIndex, links.GetTarget(usage));
705                         }
706                     }
707                 }
708                 if (usagesAsTargetCount > 0)
709                 {
710                     links.Each(usagesFiller.AddFirstAndReturnConstant,
711                       ↳ usagesAsTargetQuery);
712                     for (; i < usages.Length; i++)
713                     {
714                         var usage = usages[i];
715                         if (!equalityComparer.Equals(usage, oldLinkIndex))
716                         {
717                             links.Update(usage, links.GetSource(usage), newLinkIndex);
718                         }
719                     }
720                 }
721                 ArrayPool.Free(usages);
722             }
723         }
724     }
725     return newLinkIndex;
726 }
727
728 /// <summary>
729 /// Replace one link with another (replaced link is deleted, children are updated or
730   ↳ deleted).
731 /// </summary>
732 [MethodImpl(MethodImplOptions.AggressiveInlining)]
733 public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
734   ↳ TLink newLinkIndex)
735 {
736     var equalityComparer = EqualityComparer<TLink>.Default;
737     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
738     {
739         links.MergeUsages(oldLinkIndex, newLinkIndex);
740         links.Delete(oldLinkIndex);
741     }
742     return newLinkIndex;
743 }

```

```

733     }
734
735     public static ILinks<TLink>
736     ↪ DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
737     {
738         links = new LinksCascadeUsagesResolver<TLink>(links);
739         links = new NonNullContentsLinkDeletionResolver<TLink>(links);
740         links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
741         return links;
742     }
743 }

```

./Platform.Data.Doublets/Incrementers/FrequencyIncrementer.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.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 }

```

./Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.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.Incrementers
7  {
8      public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11         ↪ EqualityComparer<TLink>.Default;
12
13         private readonly TLink _unaryOne;
14
15         public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
16         ↪ _unaryOne = unaryOne;
17
18         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             {

```

```

26         return Links.GetOrCreate(unaryNumber, _unaryOne);
27     }
28     else
29     {
30         return Links.GetOrCreate(source, Increment(target));
31     }
32 }
33 }
34 }

```

./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  }

```

./Platform.Data.Doublets/Link.cs

```

1  using Platform.Collections.Lists;
2  using Platform.Exceptions;
3  using Platform.Ranges;
4  using Platform.Singletons;
5  using System;
6  using System.Collections;
7  using System.Collections.Generic;
8  using System.Runtime.CompilerServices;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets
13 {
14     /// <summary>
15     /// Структура описывающая уникальную связь.
16     /// </summary>
17     public struct Link<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>, IList<TLink>
18     {
19         public static readonly Link<TLink> Null = new Link<TLink>();
20
21         private static readonly LinksConstants<TLink> _constants =
22             ↳ Default<LinksConstants<TLink>>.Instance;
23         private static readonly EqualityComparer<TLink> _equalityComparer =
24             ↳ EqualityComparer<TLink>.Default;
25
26         private const int Length = 3;
27
28         public readonly TLink Index;
29         public readonly TLink Source;
30         public readonly TLink Target;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public Link(params TLink[] values) => SetValues(values, out Index, out Source, out
34             ↳ Target);
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public Link(IList<TLink> values) => SetValues(values, out Index, out Source, out Target);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public Link(object other)
41         {
42             if (other is Link<TLink> otherLink)
43             {
44                 SetValues(ref otherLink, out Index, out Source, out Target);
45             }
46             else if (other is IList<TLink> otherList)
47             {
48                 SetValues(otherList, out Index, out Source, out Target);
49             }
50             else
51             {
52                 throw new NotSupportedException();
53             }
54         }
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         public Link(ref Link<TLink> other) => SetValues(ref other, out Index, out Source, out
58             ↳ Target);
59     }
60 }

```

```

55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 public Link(TLink index, TLink source, TLink target)
57 {
58     Index = index;
59     Source = source;
60     Target = target;
61 }
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 private static void SetValues(ref Link<TLink> other, out TLink index, out TLink source,
65     ↪ out TLink target)
66 {
67     index = other.Index;
68     source = other.Source;
69     target = other.Target;
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 private static void SetValues(IList<TLink> values, out TLink index, out TLink source,
74     ↪ out TLink target)
75 {
76     switch (values.Count)
77     {
78         case 3:
79             index = values[0];
80             source = values[1];
81             target = values[2];
82             break;
83         case 2:
84             index = values[0];
85             source = values[1];
86             target = default;
87             break;
88         case 1:
89             index = values[0];
90             source = default;
91             target = default;
92             break;
93         default:
94             index = default;
95             source = default;
96             target = default;
97             break;
98     }
99 }
100
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 public override int GetHashCode() => (Index, Source, Target).GetHashCode();
103
104 [MethodImpl(MethodImplOptions.AggressiveInlining)]
105 public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
106     && _equalityComparer.Equals(Source, _constants.Null)
107     && _equalityComparer.Equals(Target, _constants.Null);
108
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 public override bool Equals(object other) => other is Link<TLink> &&
111     ↪ Equals((Link<TLink>)other);
112
113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
114 public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
115     && _equalityComparer.Equals(Source, other.Source)
116     && _equalityComparer.Equals(Target, other.Target);
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 public static string ToString(TLink index, TLink source, TLink target) => $"{index}:
120     ↪ {source}->{target}";
121
122 [MethodImpl(MethodImplOptions.AggressiveInlining)]
123 public static string ToString(TLink source, TLink target) => $"{source}->{target}";
124
125 [MethodImpl(MethodImplOptions.AggressiveInlining)]
126 public static implicit operator TLink[] (Link<TLink> link) => link.ToArray();
127
128 [MethodImpl(MethodImplOptions.AggressiveInlining)]
129 public static implicit operator Link<TLink>(TLink[] linkArray) => new
130     ↪ Link<TLink>(linkArray);
131
132 [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),
143             ↳ nameof(index));
144         if (index == _constants.IndexPart)
145         {
146             return Index;
147         }
148         if (index == _constants.SourcePart)
149         {
150             return Source;
151         }
152         if (index == _constants.TargetPart)
153         {
154             return Target;
155         }
156         throw new NotSupportedException(); // Impossible path due to
157             ↳ Ensure.ArgumentInRange
158     }
159     [MethodImpl(MethodImplOptions.AggressiveInlining)]
160     set => throw new NotSupportedException();
161 }
162
163 [MethodImpl(MethodImplOptions.AggressiveInlining)]
164 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
165
166 [MethodImpl(MethodImplOptions.AggressiveInlining)]
167 public IEnumerator<TLink> GetEnumerator()
168 {
169     yield return Index;
170     yield return Source;
171     yield return Target;
172 }
173
174 [MethodImpl(MethodImplOptions.AggressiveInlining)]
175 public void Add(TLink item) => throw new NotSupportedException();
176
177 [MethodImpl(MethodImplOptions.AggressiveInlining)]
178 public void Clear() => throw new NotSupportedException();
179
180 [MethodImpl(MethodImplOptions.AggressiveInlining)]
181 public bool Contains(TLink item) => IndexOf(item) >= 0;
182
183 [MethodImpl(MethodImplOptions.AggressiveInlining)]
184 public void CopyTo(TLink[] array, int arrayIndex)
185 {
186     Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
187     Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
188         ↳ nameof(arrayIndex));
189     if (arrayIndex + Length > array.Length)
190     {
191         throw new InvalidOperationException();
192     }
193     array[arrayIndex++] = Index;
194     array[arrayIndex++] = Source;
195     array[arrayIndex] = Target;
196 }
197
198 [MethodImpl(MethodImplOptions.AggressiveInlining)]
199 public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
200
201 [MethodImpl(MethodImplOptions.AggressiveInlining)]
202 public int IndexOf(TLink item)
203 {
204     if (_equalityComparer.Equals(Index, item))
205     {
206         return _constants.IndexPart;
207     }
208     if (_equalityComparer.Equals(Source, item))
209     {
210         return _constants.SourcePart;
211     }
212     if (_equalityComparer.Equals(Target, item))
213     {
214         return _constants.TargetPart;
215     }
216     return -1;
217 }

```

```

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

```

./Platform.Data.Doublets/LinkExtensions.cs

```

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

```

./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 }

```

./Platform.Data.Doublets/Numbers/Raw/AddressToRawNumberConverter.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.Numbers.Raw
6  {
7      public class AddressToRawNumberConverter<TLink> : IConverter<TLink>
8      {
9          public TLink Convert(TLink source) => new Hybrid<TLink>(source, isExternal: true);
10     }
11 }

```

./Platform.Data.Doublets/Numbers/Raw/RawNumberToAddressConverter.cs

```

1  using Platform.Interfaces;
2  using Platform.Numbers;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Numbers.Raw
7  {
8      public class RawNumberToAddressConverter<TLink> : IConverter<TLink>
9      {
10         public TLink Convert(TLink source) => (Integer<TLink>)new
11             ↪ Hybrid<TLink>(source).AbsoluteValue;
12     }
13 }

```

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

```
1 using System.Collections.Generic;
2 using Platform.Interfaces;
3 using Platform.Reflection;
4 using Platform.Numbers;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Numbers.Unary
9 {
10     public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
11         ↪ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↪ EqualityComparer<TLink>.Default;
15
16         private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
17
18         public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
19             ↪ powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
20             ↪ powerOf2ToUnaryNumberConverter;
21
22         public TLink Convert(TLink number)
23         {
24             var nullConstant = Links.Constants.Null;
25             var one = Integer<TLink>.One;
26             var target = nullConstant;
27             for (int i = 0; !_equalityComparer.Equals(number, default) && i <
28                 ↪ NumericType<TLink>.BitsLength; i++)
29             {
30                 if (_equalityComparer.Equals(Bit.And(number, one), one))
31                 {
32                     target = _equalityComparer.Equals(target, nullConstant)
33                         ? _powerOf2ToUnaryNumberConverter.Convert(i)
34                         : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
35                 }
36                 number = Bit.ShiftRight(number, 1);
37             }
38             return target;
39         }
40     }
41 }
```

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

```
1 using System;
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.Numbers.Unary
8 {
9     public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
10         ↪ IConverter<Doublet<TLink>, TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↪ EqualityComparer<TLink>.Default;
14
15         private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
16         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
17
18         public LinkToItsFrequencyNumberConveter(
19             ILinks<TLink> links,
20             IPropertyOperator<TLink, TLink> frequencyPropertyOperator,
21             IConverter<TLink> unaryNumberToAddressConverter)
22             : base(links)
23         {
24             _frequencyPropertyOperator = frequencyPropertyOperator;
25             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
26         }
27
28         public TLink Convert(Doublet<TLink> doublet)
29         {
30             var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
31             if (_equalityComparer.Equals(link, default))
32             {
33                 throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
34             }
35             var frequency = _frequencyPropertyOperator.Get(link);
36             if (_equalityComparer.Equals(frequency, default))
37             {
38                 throw new ArgumentException($"Frequency of {link} not found.", nameof(link));
39             }
40             return _unaryNumberToAddressConverter.Convert(frequency);
41         }
42     }
43 }
```

```

35         {
36             return default;
37         }
38         var frequencyNumber = Links.GetSource(frequency);
39         return _unaryNumberToAddressConverter.Convert(frequencyNumber);
40     }
41 }
42 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Exceptions;
3  using Platform.Interfaces;
4  using Platform.Ranges;
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 }

```

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

```

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

```



```

29         { _unaryOne, one }
30     };
31     var unary = _unaryOne;
32     var number = one;
33     for (var i = 1; i < 64; i++)
34     {
35         unary = Links.GetOrCreate(unary, unary);
36         number = Double(number);
37         _unaryToUInt64.Add(unary, number);
38     }
39 }
40
41 public TLink Convert(TLink unaryNumber)
42 {
43     if (_equalityComparer.Equals(unaryNumber, default))
44     {
45         return default;
46     }
47     if (_equalityComparer.Equals(unaryNumber, _unaryOne))
48     {
49         return Integer<TLink>.One;
50     }
51     var source = Links.GetSource(unaryNumber);
52     var target = Links.GetTarget(unaryNumber);
53     if (_equalityComparer.Equals(source, target))
54     {
55         return _unaryToUInt64[unaryNumber];
56     }
57     else
58     {
59         var result = _unaryToUInt64[source];
60         TLink lastValue;
61         while (!_unaryToUInt64.TryGetValue(target, out lastValue))
62         {
63             source = Links.GetSource(target);
64             result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
65             target = Links.GetTarget(target);
66         }
67         result = Arithmetic<TLink>.Add(result, lastValue);
68         return result;
69     }
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 private static TLink Double(TLink number) => (Integer<TLink>)((Integer<TLink>)number *
74     ↪ 2UL);
75 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Interfaces;
4 using Platform.Reflection;
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>.BitsLength; i++)
25             {
26                 _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
27             }
28         }
29     }
30 }

```

```

27     public TLink Convert(TLink sourceNumber)
28     {
29         var nullConstant = Links.Constants.Null;
30         var source = sourceNumber;
31         var target = nullConstant;
32         if (!_equalityComparer.Equals(source, nullConstant))
33         {
34             while (true)
35             {
36                 if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
37                 {
38                     SetBit(ref target, powerOf2Index);
39                     break;
40                 }
41                 else
42                 {
43                     powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
44                     SetBit(ref target, powerOf2Index);
45                     source = Links.GetTarget(source);
46                 }
47             }
48         }
49         return target;
50     }
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     private static void SetBit(ref TLink target, int powerOf2Index) => target =
54         ↪ Bit.Or(target, Bit.ShiftLeft(Integer<TLink>.One, powerOf2Index));
55 }

```

./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>,
10         ↪ IPropertiesOperator<TLink, TLink, TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↪ EqualityComparer<TLink>.Default;
14
15         public PropertiesOperator(ILinks<TLink> links) : base(links) { }
16
17         public TLink GetValue(TLink @object, TLink property)
18         {
19             var objectProperty = Links.SearchOrDefault(@object, property);
20             if (_equalityComparer.Equals(objectProperty, default))
21             {
22                 return default;
23             }
24             var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
25             if (valueLink == null)
26             {
27                 return default;
28             }
29             return Links.GetTarget(valueLink[Links.Constants.IndexPart]);
30         }
31
32         public void SetValue(TLink @object, TLink property, TLink value)
33         {
34             var objectProperty = Links.GetOrCreate(@object, property);
35             Links.DeleteMany(Links.AllIndices(Links.Constants.Any, objectProperty));
36             Links.GetOrCreate(objectProperty, value);
37         }
38     }
39 }

```

./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>, IPropertyOperator<TLink,
9         ↪ TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13
14         private readonly TLink _propertyMarker;
15         private readonly TLink _propertyValueMarker;
16
17         public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
18             ↪ propertyValueMarker) : base(links)
19         {
20             _propertyMarker = propertyMarker;
21             _propertyValueMarker = propertyValueMarker;
22         }
23
24         public TLink Get(TLink link)
25         {
26             var property = Links.SearchOrDefault(link, _propertyMarker);
27             var container = GetContainer(property);
28             var value = GetValue(container);
29             return value;
30         }
31
32         private TLink GetContainer(TLink property)
33         {
34             var valueContainer = default(TLink);
35             if (_equalityComparer.Equals(property, default))
36             {
37                 return valueContainer;
38             }
39             var constants = Links.Constants;
40             var countinueConstant = constants.Continue;
41             var breakConstant = constants.Break;
42             var anyConstant = constants.Any;
43             var query = new Link<TLink>(anyConstant, property, anyConstant);
44             Links.Each(candidate =>
45             {
46                 var candidateTarget = Links.GetTarget(candidate);
47                 var valueTarget = Links.GetTarget(candidateTarget);
48                 if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
49                 {
50                     valueContainer = Links.GetIndex(candidate);
51                     return breakConstant;
52                 }
53                 return countinueConstant;
54             }, query);
55             return valueContainer;
56         }
57
58         private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
59             ↪ ? default : Links.GetTarget(container);
60
61         public void Set(TLink link, TLink value)
62         {
63             var property = Links.GetOrCreate(link, _propertyMarker);
64             var container = GetContainer(property);
65             if (_equalityComparer.Equals(container, default))
66             {
67                 Links.GetOrCreate(property, value);
68             }
69             else
70             {
71                 Links.Update(container, property, value);
72             }
73         }
74     }
75 }

```

./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.Numbers;
6 using Platform.Collections.Methods.Trees;
7 using static System.Runtime.CompilerServices.Unsafe;
8
9 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

```

```

10
11 namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
12 {
13     public unsafe abstract class LinksAvlBalancedTreeMethodsBase<TLink> :
14         ↳ SizedAndThreadedAVLBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
15     {
16         protected readonly TLink Break;
17         protected readonly TLink Continue;
18         protected readonly byte* Links;
19         protected readonly byte* Header;
20
21         public LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
22             ↳ byte* header)
23         {
24             Links = links;
25             Header = header;
26             Break = constants.Break;
27             Continue = constants.Continue;
28
29             [MethodImpl(MethodImplOptions.AggressiveInlining)]
30             protected abstract TLink GetTreeRoot();
31
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             protected abstract TLink GetBasePartValue(TLink link);
34
35             [MethodImpl(MethodImplOptions.AggressiveInlining)]
36             protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
37                 ↳ rootSource, TLink rootTarget);
38
39             [MethodImpl(MethodImplOptions.AggressiveInlining)]
40             protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
41                 ↳ rootSource, TLink rootTarget);
42
43             [MethodImpl(MethodImplOptions.AggressiveInlining)]
44             protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
45                 ↳ AsRef<LinksHeader<TLink>>(Header);
46
47             [MethodImpl(MethodImplOptions.AggressiveInlining)]
48             protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
49                 ↳ AsRef<RawLink<TLink>>(Links + RawLink<TLink>.SizeInBytes * (Integer<TLink>)link);
50
51             [MethodImpl(MethodImplOptions.AggressiveInlining)]
52             protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
53             {
54                 ref var link = ref GetLinkReference(linkIndex);
55                 return new Link<TLink>(linkIndex, link.Source, link.Target);
56             }
57
58             [MethodImpl(MethodImplOptions.AggressiveInlining)]
59             protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
60             {
61                 ref var firstLink = ref GetLinkReference(first);
62                 ref var secondLink = ref GetLinkReference(second);
63                 return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
64                     ↳ secondLink.Source, secondLink.Target);
65             }
66
67             [MethodImpl(MethodImplOptions.AggressiveInlining)]
68             protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
69             {
70                 ref var firstLink = ref GetLinkReference(first);
71                 ref var secondLink = ref GetLinkReference(second);
72                 return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
73                     ↳ secondLink.Source, secondLink.Target);
74             }
75
76             [MethodImpl(MethodImplOptions.AggressiveInlining)]
77             protected virtual TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5,
78                 ↳ -5);
79
80             [MethodImpl(MethodImplOptions.AggressiveInlining)]
81             protected virtual void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
82                 ↳ Bit<TLink>.PartialWrite(storedValue, size, 5, -5);
83
84             [MethodImpl(MethodImplOptions.AggressiveInlining)]
85             protected virtual bool GetLeftIsChildValue(TLink value)
86             {
87                 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
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 protected virtual bool GetRightIsChildValue(TLink value)
99 {
100     unchecked
101     {
102         //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 3, 1);
103         return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
104     }
105 }
106
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]
108 protected virtual void SetRightIsChildValue(ref TLink storedValue, bool value)
109 {
110     unchecked
111     {
112         var previousValue = storedValue;
113         var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 3,
114             ↪ 1);
115         storedValue = modified;
116     }
117 }
118
119 [MethodImpl(MethodImplOptions.AggressiveInlining)]
120 protected virtual sbyte GetBalanceValue(TLink storedValue)
121 {
122     unchecked
123     {
124         var value = (int)(Integer<TLink>)Bit<TLink>.PartialRead(storedValue, 0, 3);
125         value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
126             ↪ end of sbyte
127         return (sbyte)value;
128     }
129 }
130
131 [MethodImpl(MethodImplOptions.AggressiveInlining)]
132 protected virtual void SetBalanceValue(ref TLink storedValue, sbyte value)
133 {
134     unchecked
135     {
136         var packagedValue = (TLink)(Integer<TLink>)((byte)value >> 5 & 4 | value & 3);
137         var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
138         storedValue = modified;
139     }
140 }
141
142 public TLink this[TLink index]
143 {
144     get
145     {
146         var root = GetTreeRoot();
147         if (GreaterOrEqualThan(index, GetSize(root)))
148         {
149             return Zero;
150         }
151         while (!EqualToZero(root))
152         {
153             var left = GetLeftOrDefault(root);
154             var leftSize = GetSizeOrZero(left);
155             if (LessThan(index, leftSize))
156             {
157                 root = left;
158             }
159         }
160     }
161 }

```

```

155         continue;
156     }
157     if (IsEquals(index, leftSize))
158     {
159         return root;
160     }
161     root = GetRightOrDefault(root);
162     index = Subtract(index, Increment(leftSize));
163 }
164 return Zero; // TODO: Impossible situation exception (only if tree structure
    ↳ broken)
165 }
166 }
167
168 /// <summary>
169 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
    ↳ (концом).
170 /// </summary>
171 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
172 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
173 /// <returns>Индекс искомой связи.</returns>
174 public TLink Search(TLink source, TLink target)
175 {
176     var root = GetTreeRoot();
177     while (!EqualToZero(root))
178     {
179         ref var rootLink = ref GetLinkReference(root);
180         var rootSource = rootLink.Source;
181         var rootTarget = rootLink.Target;
182         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
            ↳ node.Key < root.Key
183         {
184             root = GetLeftOrDefault(root);
185         }
186         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
            ↳ node.Key > root.Key
187         {
188             root = GetRightOrDefault(root);
189         }
190         else // node.Key == root.Key
191         {
192             return root;
193         }
194     }
195     return Zero;
196 }
197
198 // TODO: Return indices range instead of references count
199 public TLink CountUsages(TLink link)
200 {
201     var root = GetTreeRoot();
202     var total = GetSize(root);
203     var totalRightIgnore = Zero;
204     while (!EqualToZero(root))
205     {
206         var @base = GetBasePartValue(root);
207         if (LessOrEqualThan(@base, link))
208         {
209             root = GetRightOrDefault(root);
210         }
211         else
212         {
213             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
214             root = GetLeftOrDefault(root);
215         }
216     }
217     root = GetTreeRoot();
218     var totalLeftIgnore = Zero;
219     while (!EqualToZero(root))
220     {
221         var @base = GetBasePartValue(root);
222         if (GreaterOrEqualThan(@base, link))
223         {
224             root = GetLeftOrDefault(root);
225         }
226         else
227         {
228             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));

```

```

229         root = GetRightOrDefault(root);
230     }
231 }
232 }
233 return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
234 }
235
236 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
237 {
238     var root = GetTreeRoot();
239     if (EqualToZero(root))
240     {
241         return Continue;
242     }
243     TLink first = Zero, current = root;
244     while (!EqualToZero(current))
245     {
246         var @base = GetBasePartValue(current);
247         if (GreaterOrEqualThan(@base, link))
248         {
249             if (IsEquals(@base, link))
250             {
251                 first = current;
252             }
253             current = GetLeftOrDefault(current);
254         }
255         else
256         {
257             current = GetRightOrDefault(current);
258         }
259     }
260     if (!EqualToZero(first))
261     {
262         current = first;
263         while (true)
264         {
265             if (IsEquals(handler(GetLinkValues(current)), Break))
266             {
267                 return Break;
268             }
269             current = GetNext(current);
270             if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
271             {
272                 break;
273             }
274         }
275     }
276     return Continue;
277 }
278
279 protected override void PrintNodeValue(TLink node, StringBuilder sb)
280 {
281     ref var link = ref GetLinkReference(node);
282     sb.Append(' ');
283     sb.Append(link.Source);
284     sb.Append('-');
285     sb.Append('>');
286     sb.Append(link.Target);
287 }
288 }
289 }

```

./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.Numbers;
6 using Platform.Collections.Methods.Trees;
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     ↪ SizeBalancedTreeMethods2<TLink>, ILinksTreeMethods<TLink>
15     {
16         protected readonly TLink Break;
17         protected readonly TLink Continue;

```

```

17     protected readonly byte* Links;
18     protected readonly byte* Header;
19
20     public LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
21     ↪ byte* header)
22     {
23         Links = links;
24         Header = header;
25         Break = constants.Break;
26         Continue = constants.Continue;
27     }
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     protected abstract TLink GetTreeRoot();
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     protected abstract TLink GetBasePartValue(TLink link);
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
37     ↪ rootSource, TLink rootTarget);
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
41     ↪ rootSource, TLink rootTarget);
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
45     ↪ AsRef<LinksHeader<TLink>>(Header);
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
49     ↪ AsRef<RawLink<TLink>>(Links + RawLink<TLink>.SizeInBytes * (Integer<TLink>)link);
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
53     {
54         ref var link = ref GetLinkReference(linkIndex);
55         return new Link<TLink>(linkIndex, link.Source, link.Target);
56     }
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
60     {
61         ref var firstLink = ref GetLinkReference(first);
62         ref var secondLink = ref GetLinkReference(second);
63         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
64         ↪ secondLink.Source, secondLink.Target);
65     }
66
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
69     {
70         ref var firstLink = ref GetLinkReference(first);
71         ref var secondLink = ref GetLinkReference(second);
72         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
73         ↪ secondLink.Source, secondLink.Target);
74     }
75
76     public TLink this[TLink index]
77     {
78         get
79         {
80             var root = GetTreeRoot();
81             if (GreaterOrEqualThan(index, GetSize(root)))
82             {
83                 return Zero;
84             }
85             while (!EqualToZero(root))
86             {
87                 var left = GetLeftOrDefault(root);
88                 var leftSize = GetSizeOrZero(left);
89                 if (LessThan(index, leftSize))
90                 {
91                     root = left;
92                     continue;
93                 }
94                 if (IsEquals(index, leftSize))
95                 {

```



```

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) =>
168         ↳ EachUsageCore(@base, GetTreeRoot(), handler);
169
170     // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
171     ↳ low-level MSIL stack.
172     private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
173     {
174         var @continue = Continue;
175         if (EqualToZero(link))
176         {
177             return @continue;
178         }
179         var linkBasePart = GetBasePartValue(link);
180         var @break = Break;
181         if (GreaterThan(linkBasePart, @base))
182         {
183             if (IsEquals(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
184             {
185                 return @break;
186             }
187         }
188         else if (LessThan(linkBasePart, @base))
189         {
190             if (IsEquals(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
191             {
192                 return @break;
193             }
194         }
195         else //if (linkBasePart == @base)
196         {
197             if (IsEquals(handler(GetLinkValues(link)), @break))
198             {
199                 return @break;
200             }
201             if (IsEquals(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
202             {
203                 return @break;
204             }
205             if (IsEquals(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
206             {
207                 return @break;
208             }
209         }
210         return @continue;
211     }
212
213     protected override void PrintNodeValue(TLink node, StringBuilder sb)
214     {
215         ref var link = ref GetLinkReference(node);
216         sb.Append(' ');
217         sb.Append(link.Source);
218         sb.Append('-');
219         sb.Append('>');
220         sb.Append(link.Target);
221     }
222 }

```

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

```

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

```

```

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

```

./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             ↳ IsEquals(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             ↳ IsEquals(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 }
```

./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
11 }
```

```

9 public LinksTargetsAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
10     ↳ byte* header) : base(constants, links, header) { }
11
12 [MethodImpl(MethodImplOptions.AggressiveInlining)]
13 protected unsafe override ref TLink GetLeftReference(TLink node) => ref
14     ↳ GetLinkReference(node).LeftAsTarget;
15
16 [MethodImpl(MethodImplOptions.AggressiveInlining)]
17 protected unsafe override ref TLink GetRightReference(TLink node) => ref
18     ↳ GetLinkReference(node).RightAsTarget;
19
20 [MethodImpl(MethodImplOptions.AggressiveInlining)]
21 protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
22
23 [MethodImpl(MethodImplOptions.AggressiveInlining)]
24 protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
25
26 [MethodImpl(MethodImplOptions.AggressiveInlining)]
27 protected override void SetLeft(TLink node, TLink left) =>
28     ↳ GetLinkReference(node).LeftAsTarget = left;
29
30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 protected override void SetRight(TLink node, TLink right) =>
32     ↳ GetLinkReference(node).RightAsTarget = right;
33
34 [MethodImpl(MethodImplOptions.AggressiveInlining)]
35 protected override TLink GetSize(TLink node) =>
36     ↳ GetSizeValue(GetLinkReference(node).SizeAsTarget);
37
38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
40     ↳ GetLinkReference(node).SizeAsTarget, size);
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected override bool GetLeftIsChild(TLink node) =>
44     ↳ GetLeftIsChildValue(GetLinkReference(node).SizeAsTarget);
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 protected override void SetLeftIsChild(TLink node, bool value) =>
48     ↳ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 protected override bool GetRightIsChild(TLink node) =>
52     ↳ GetRightIsChildValue(GetLinkReference(node).SizeAsTarget);
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected override void SetRightIsChild(TLink node, bool value) =>
56     ↳ SetRightIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 protected override sbyte GetBalance(TLink node) =>
60     ↳ GetBalanceValue(GetLinkReference(node).SizeAsTarget);
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
64     ↳ GetLinkReference(node).SizeAsTarget, value);
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsTarget;
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
74     ↳ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
75     ↳ IsEquals(firstTarget, secondTarget) && LessThan(firstSource, secondSource);
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
79     ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
80     ↳ IsEquals(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource);
81
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 protected override void ClearNode(TLink node)
84 {
85     ref var link = ref GetLinkReference(node);
86     link.LeftAsTarget = Zero;
87 }

```

```

70         link.RightAsTarget = Zero;
71         link.SizeAsTarget = Zero;
72     }
73 }
74 }

```

./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             ↳ IsEquals(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             ↳ IsEquals(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 }

```

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

```

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

```

```

4 using Platform.Collections.Arrays;
5 using Platform.Data.Exceptions;
6 using Platform.Disposables;
7 using Platform.Memory;
8 using Platform.Numbers;
9 using Platform.Singletons;
10
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
14 {
15     public abstract class ResizableDirectMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16     {
17         protected static readonly EqualityComparer<TLink> EqualityComparer =
18             ↳ EqualityComparer<TLink>.Default;
19         protected static readonly Comparer<TLink> Comparer = Comparer<TLink>.Default;
20
21         /// <summary>Возвращает размер одной связи в байтах.</summary>
22         /// <remarks>
23         /// Используется только во вне класса, не рекомендуется использовать внутри.
24         /// Так как во вне не обязательно будет доступен unsafe C#.
25         /// </remarks>
26         public static readonly long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;
27
28         public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
29
30         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
31
32         protected readonly IResizableDirectMemory _memory;
33         protected readonly long _memoryReservationStep;
34
35         protected ILinksTreeMethods<TLink> TargetsTreeMethods;
36         protected ILinksTreeMethods<TLink> SourcesTreeMethods;
37         // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
38         // ↳ нужно использовать не список а дерево, так как так можно быстрее проверить на
39         // ↳ наличие связи внутри
40         protected ILinksListMethods<TLink> UnusedLinksListMethods;
41
42         /// <summary>
43         /// Возвращает общее число связей находящихся в хранилище.
44         /// </summary>
45         protected virtual TLink Total
46         {
47             get
48             {
49                 ref var header = ref GetHeaderReference();
50                 return Subtract(header.AllocatedLinks, header.FreeLinks);
51             }
52         }
53
54         public virtual LinksConstants<TLink> Constants { get; }
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         public ResizableDirectMemoryLinksBase(IResizableDirectMemory memory, long
58             ↳ memoryReservationStep, LinksConstants<TLink> constants)
59         {
60             _memory = memory;
61             _memoryReservationStep = memoryReservationStep;
62             Constants = constants;
63         }
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         public ResizableDirectMemoryLinksBase(IResizableDirectMemory memory, long
67             ↳ memoryReservationStep) : this(memory, memoryReservationStep,
68             ↳ Default<LinksConstants<TLink>>.Instance) { }
69
70         protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
71         {
72             if (memory.ReservedCapacity < memoryReservationStep)
73             {
74                 memory.ReservedCapacity = memoryReservationStep;
75             }
76             SetPointers(_memory);
77             ref var header = ref GetHeaderReference();
78             // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
79             _memory.UsedCapacity = ConvertToUInt64(header.AllocatedLinks) * LinkSizeInBytes +
80                 ↳ LinkHeaderSizeInBytes;
81             // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
82             header.ReservedLinks = ConvertToAddress((_memory.ReservedCapacity -
83                 ↳ LinkHeaderSizeInBytes) / LinkSizeInBytes);
84         }
85     }
86 }

```

```

76     }
77
78     [MethodImpl(MethodImplOptions.AggressiveInlining)]
79     public virtual TLink Count(IList<TLink> restrictions)
80     {
81         // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
82         if (restrictions.Count == 0)
83         {
84             return Total;
85         }
86         var constants = Constants;
87         var any = constants.Any;
88         var index = restrictions[constants.IndexPart];
89         if (restrictions.Count == 1)
90         {
91             if (AreEqual(index, any))
92             {
93                 return Total;
94             }
95             return Exists(index) ? GetOne() : GetZero();
96         }
97         if (restrictions.Count == 2)
98         {
99             var value = restrictions[1];
100             if (AreEqual(index, any))
101             {
102                 if (AreEqual(value, any))
103                 {
104                     return Total; // Any - как отсутствие ограничения
105                 }
106                 return Add(SourcesTreeMethods.CountUsages(value),
107                     ↪ TargetsTreeMethods.CountUsages(value));
108             }
109             else
110             {
111                 if (!Exists(index))
112                 {
113                     return GetZero();
114                 }
115                 if (AreEqual(value, any))
116                 {
117                     return GetOne();
118                 }
119                 ref var storedLinkValue = ref GetLinkReference(index);
120                 if (AreEqual(storedLinkValue.Source, value) ||
121                     ↪ AreEqual(storedLinkValue.Target, value))
122                 {
123                     return GetOne();
124                 }
125                 return GetZero();
126             }
127         }
128         if (restrictions.Count == 3)
129         {
130             var source = restrictions[constants.SourcePart];
131             var target = restrictions[constants.TargetPart];
132             if (AreEqual(index, any))
133             {
134                 if (AreEqual(source, any) && AreEqual(target, any))
135                 {
136                     return Total;
137                 }
138                 else if (AreEqual(source, any))
139                 {
140                     return TargetsTreeMethods.CountUsages(target);
141                 }
142                 else if (AreEqual(target, any))
143                 {
144                     return SourcesTreeMethods.CountUsages(source);
145                 }
146                 else //if(source != Any && target != Any)
147                 {
148                     // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
149                     var link = SourcesTreeMethods.Search(source, target);
150                     return AreEqual(link, constants.Null) ? GetZero() : GetOne();
151                 }
152             }
153             else

```



```

152     {
153         if (!Exists(index))
154         {
155             return GetZero();
156         }
157         if (AreEqual(source, any) && AreEqual(target, any))
158         {
159             return GetOne();
160         }
161         ref var storedLinkValue = ref GetLinkReference(index);
162         if (!AreEqual(source, any) && !AreEqual(target, any))
163         {
164             if (AreEqual(storedLinkValue.Source, source) &&
165                 ↪ AreEqual(storedLinkValue.Target, target))
166             {
167                 return GetOne();
168             }
169             return GetZero();
170         }
171         var value = default(TLink);
172         if (AreEqual(source, any))
173         {
174             value = target;
175         }
176         if (AreEqual(target, any))
177         {
178             value = source;
179         }
180         if (AreEqual(storedLinkValue.Source, value) ||
181             ↪ AreEqual(storedLinkValue.Target, value))
182         {
183             return GetOne();
184         }
185         return GetZero();
186     }
187 }
188
189 [MethodImpl(MethodImplOptions.AggressiveInlining)]
190 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
191 {
192     var constants = Constants;
193     var @break = constants.Break;
194     if (restrictions.Count == 0)
195     {
196         for (var link = GetOne(); LessOrEqualThan(link,
197             ↪ GetHeaderReference().AllocatedLinks); link = Increment(link))
198         {
199             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
200             {
201                 return @break;
202             }
203         }
204         return @break;
205     }
206     var @continue = constants.Continue;
207     var any = constants.Any;
208     var index = restrictions[constants.IndexPart];
209     if (restrictions.Count == 1)
210     {
211         if (AreEqual(index, any))
212         {
213             return Each(handler, GetEmptyList());
214         }
215         if (!Exists(index))
216         {
217             return @continue;
218         }
219         return handler(GetLinkStruct(index));
220     }
221     if (restrictions.Count == 2)
222     {
223         var value = restrictions[1];
224         if (AreEqual(index, any))
225         {
226             if (AreEqual(value, any))

```

```

226         {
227             return Each(handler, GetEmptyList());
228         }
229         if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
230         {
231             return @break;
232         }
233         return Each(handler, new Link<TLink>(index, any, value));
234     }
235     else
236     {
237         if (!Exists(index))
238         {
239             return @continue;
240         }
241         if (AreEqual(value, any))
242         {
243             return handler(GetLinkStruct(index));
244         }
245         ref var storedLinkValue = ref GetLinkReference(index);
246         if (AreEqual(storedLinkValue.Source, value) ||
247             AreEqual(storedLinkValue.Target, value))
248         {
249             return handler(GetLinkStruct(index));
250         }
251         return @continue;
252     }
253 }
254 if (restrictions.Count == 3)
255 {
256     var source = restrictions[constants.SourcePart];
257     var target = restrictions[constants.TargetPart];
258     if (AreEqual(index, any))
259     {
260         if (AreEqual(source, any) && AreEqual(target, any))
261         {
262             return Each(handler, GetEmptyList());
263         }
264         else if (AreEqual(source, any))
265         {
266             return TargetsTreeMethods.EachUsage(target, handler);
267         }
268         else if (AreEqual(target, any))
269         {
270             return SourcesTreeMethods.EachUsage(source, handler);
271         }
272         else //if(source != Any && target != Any)
273         {
274             var link = SourcesTreeMethods.Search(source, target);
275             return AreEqual(link, constants.Null) ? @continue :
276                 ↪ handler(GetLinkStruct(link));
277         }
278     }
279     else
280     {
281         if (!Exists(index))
282         {
283             return @continue;
284         }
285         if (AreEqual(source, any) && AreEqual(target, any))
286         {
287             return handler(GetLinkStruct(index));
288         }
289         ref var storedLinkValue = ref GetLinkReference(index);
290         if (!AreEqual(source, any) && !AreEqual(target, any))
291         {
292             if (AreEqual(storedLinkValue.Source, source) &&
293                 AreEqual(storedLinkValue.Target, target))
294             {
295                 return handler(GetLinkStruct(index));
296             }
297             return @continue;
298         }
299         var value = default(TLink);
300         if (AreEqual(source, any))
301         {
302             value = target;

```

```

303         if (AreEqual(target, any))
304         {
305             value = source;
306         }
307         if (AreEqual(storedLinkValue.Source, value) ||
308             AreEqual(storedLinkValue.Target, value))
309         {
310             return handler(GetLinkStruct(index));
311         }
312         return @continue;
313     }
314 }
315 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
316 }
317
318 /// <remarks>
319 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↳ в другом месте (но не в менеджере памяти, а в логике Links)
320 /// </remarks>
321 [MethodImpl(MethodImplOptions.AggressiveInlining)]
322 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
323 {
324     var constants = Constants;
325     var @null = constants.Null;
326     var linkIndex = restrictions[constants.IndexPart];
327     ref var link = ref GetLinkReference(linkIndex);
328     ref var header = ref GetHeaderReference();
329     ref var firstAsSource = ref header.FirstAsSource;
330     ref var firstAsTarget = ref header.FirstAsTarget;
331     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↳ предварительно заполнено нулями
332     if (!AreEqual(link.Source, @null))
333     {
334         SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
335     }
336     if (!AreEqual(link.Target, @null))
337     {
338         TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
339     }
340     link.Source = substitution[constants.SourcePart];
341     link.Target = substitution[constants.TargetPart];
342     if (!AreEqual(link.Source, @null))
343     {
344         SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
345     }
346     if (!AreEqual(link.Target, @null))
347     {
348         TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
349     }
350     return linkIndex;
351 }
352
353 /// <remarks>
354 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
    ↳ пространство
355 /// </remarks>
356 public virtual TLink Create(IList<TLink> restrictions)
357 {
358     ref var header = ref GetHeaderReference();
359     var freeLink = header.FirstFreeLink;
360     if (!AreEqual(freeLink, Constants.Null))
361     {
362         UnusedLinksListMethods.Detach(freeLink);
363     }
364     else
365     {
366         var maximumPossibleInnerReference =
            ↳ Constants.PossibleInnerReferencesRange.Maximum;
367         if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
368         {
369             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
370         }
371         if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
372         {
373             _memory.ReservedCapacity += _memory.ReservationStep;
374             SetPointers(_memory);
375             header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
                ↳ LinkSizeInBytes);

```

```

376     }
377     header.AllocatedLinks = Increment(header.AllocatedLinks);
378     _memory.UsedCapacity += LinkSizeInBytes;
379     freeLink = header.AllocatedLinks;
380 }
381 return freeLink;
382 }
383
384 [MethodImpl(MethodImplOptions.AggressiveInlining)]
385 public virtual void Delete(ICollection<TLink> restrictions)
386 {
387     ref var header = ref GetHeaderReference();
388     var link = restrictions[Constants.IndexPart];
389     if (LessThan(link, header.AllocatedLinks))
390     {
391         UnusedLinksListMethods.AttachAsFirst(link);
392     }
393     else if (AreEqual(link, header.AllocatedLinks))
394     {
395         header.AllocatedLinks = Decrement(header.AllocatedLinks);
396         _memory.UsedCapacity -= LinkSizeInBytes;
397         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
398         //   ↳ пока не дойдём до первой существующей связи
399         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
400         while (GreaterThan(header.AllocatedLinks, GetZero()) &&
401             ↳ IsUnusedLink(header.AllocatedLinks))
402         {
403             UnusedLinksListMethods.Detach(header.AllocatedLinks);
404             header.AllocatedLinks = Decrement(header.AllocatedLinks);
405             _memory.UsedCapacity -= LinkSizeInBytes;
406         }
407     }
408 }
409
410 [MethodImpl(MethodImplOptions.AggressiveInlining)]
411 public ICollection<TLink> GetLinkStruct(TLink linkIndex)
412 {
413     ref var link = ref GetLinkReference(linkIndex);
414     return new Link<TLink>(linkIndex, link.Source, link.Target);
415 }
416
417 /// <remarks>
418 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
419 ///   ↳ адрес реально поменялся
420 ///
421 /// Указатель this.links может быть в том же месте,
422 /// так как 0-я связь не используется и имеет такой же размер как Header,
423 /// поэтому header размещается в том же месте, что и 0-я связь
424 /// </remarks>
425 [MethodImpl(MethodImplOptions.AggressiveInlining)]
426 protected abstract void SetPointers(IResizableDirectMemory memory);
427
428 [MethodImpl(MethodImplOptions.AggressiveInlining)]
429 protected virtual void ResetPointers()
430 {
431     SourcesTreeMethods = null;
432     TargetsTreeMethods = null;
433     UnusedLinksListMethods = null;
434 }
435
436 [MethodImpl(MethodImplOptions.AggressiveInlining)]
437 protected abstract ref LinkHeader<TLink> GetHeaderReference();
438
439 [MethodImpl(MethodImplOptions.AggressiveInlining)]
440 protected abstract ref RawLink<TLink> GetLinkReference(TLink linkIndex);
441
442 [MethodImpl(MethodImplOptions.AggressiveInlining)]
443 protected virtual bool Exists(TLink link)
444 => GreaterOrEqualThan(link, Constants.PossibleInnerReferencesRange.Minimum)
445     && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
446     && !IsUnusedLink(link);
447
448 [MethodImpl(MethodImplOptions.AggressiveInlining)]
449 protected virtual bool IsUnusedLink(TLink linkIndex)
450 {
451     if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
452         ↳ is not needed
453     {
454         ref var link = ref GetLinkReference(linkIndex);

```

```

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 long ConvertToUInt64(TLink value) => (Integer<TLink>)value;
483
484 [MethodImpl(MethodImplOptions.AggressiveInlining)]
485 protected virtual TLink ConvertToAddress(long value) => (Integer<TLink>)value;
486
487 [MethodImpl(MethodImplOptions.AggressiveInlining)]
488 protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
489     ↪ second);
490
491 [MethodImpl(MethodImplOptions.AggressiveInlining)]
492 protected virtual TLink Subtract(TLink first, TLink second) =>
493     ↪ Arithmetic<TLink>.Subtract(first, second);
494
495 [MethodImpl(MethodImplOptions.AggressiveInlining)]
496 protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
497
498 [MethodImpl(MethodImplOptions.AggressiveInlining)]
499 protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
500
501 [MethodImpl(MethodImplOptions.AggressiveInlining)]
502 protected virtual IList<TLink> GetEmptyList() => ArrayPool<TLink>.Empty;
503
504 #region Disposable
505
506 protected override bool AllowMultipleDisposeCalls => true;
507
508 protected override void Dispose(bool manual, bool wasDisposed)
509 {
510     if (!wasDisposed)
511     {
512         ResetPointers();
513         _memory.DisposeIfPossible();
514     }
515 }
516
517 #endregion
518 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Numbers;
3 using Platform.Memory;
4 using static System.Runtime.CompilerServices.Unsafe;

```

```

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

```

```

68     }
69
70     [MethodImpl(MethodImplOptions.AggressiveInlining)]
71     protected override ref LinksHeader<TLink> GetHeaderReference() => ref
        ↳ AsRef<LinksHeader<TLink>>(_header);
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     protected override ref RawLink<TLink> GetLinkReference(TLink linkIndex) => ref
        ↳ AsRef<RawLink<TLink>>(_links + LinkSizeInBytes * (Integer<TLink>)linkIndex);
75 }
76 }

```

./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>,
        ↳ ILinksListMethods<TLink>
11     {
12         private readonly byte* _links;
13         private readonly byte* _header;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public UnusedLinksListMethods(byte* links, byte* header)
17         {
18             _links = links;
19             _header = header;
20         }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
            ↳ AsRef<LinksHeader<TLink>>(_header);
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
            ↳ AsRef<RawLink<TLink>>(_links + RawLink<TLink>.SizeInBytes * (Integer<TLink>)link);
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected override TLink GetPrevious(TLink element) => GetLinkReference(element).Source;
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected override TLink GetNext(TLink element) => GetLinkReference(element).Target;
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override TLink GetSize() => GetHeaderReference().FreeLinks;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
            ↳ element;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
            ↳ element;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override void SetPrevious(TLink element, TLink previous) =>
            ↳ GetLinkReference(element).Source = previous;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override void SetNext(TLink element, TLink next) =>
            ↳ GetLinkReference(element).Target = next;
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
57     }
58 }

```

./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 }
```

./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 }
```

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

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

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

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

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

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



```

6
7 namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
8 {
9     public unsafe abstract class UInt64LinksAvlBalancedTreeMethodsBase :
10         ↳ LinksAvlBalancedTreeMethodsBase<ulong>
11     {
12         protected new readonly RawLink<ulong>* Links;
13         protected new readonly LinksHeader<ulong>* Header;
14
15         public UInt64LinksAvlBalancedTreeMethodsBase(LinksConstants<ulong> constants,
16             ↳ RawLink<ulong>* links, LinksHeader<ulong>* header)
17             : base(constants, (byte*)links, (byte*)header)
18         {
19             Links = links;
20             Header = header;
21
22             [MethodImpl(MethodImplOptions.AggressiveInlining)]
23             protected override ulong GetZero() => 0UL;
24
25             [MethodImpl(MethodImplOptions.AggressiveInlining)]
26             protected override bool EqualToZero(ulong value) => value == 0UL;
27
28             [MethodImpl(MethodImplOptions.AggressiveInlining)]
29             protected override bool IsEquals(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             [MethodImpl(MethodImplOptions.AggressiveInlining)]
65             protected override ulong Add(ulong first, ulong second) => first + second;
66
67             [MethodImpl(MethodImplOptions.AggressiveInlining)]
68             protected override ulong Subtract(ulong first, ulong second) => first - second;
69
70             [MethodImpl(MethodImplOptions.AggressiveInlining)]
71             protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
72             {
73                 ref var firstLink = ref Links[first];
74                 ref var secondLink = ref Links[second];
75                 return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
76                     ↳ secondLink.Source, secondLink.Target);
77             }
78
79             [MethodImpl(MethodImplOptions.AggressiveInlining)]
80             protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
81             {
82                 ref var firstLink = ref Links[first];
83                 ref var secondLink = ref Links[second];

```

```

79         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
80             ↪ secondLink.Source, secondLink.Target);
81     }
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     protected override ulong GetSizeValue(ulong value) => unchecked((value & 4294967264UL)
84         ↪ >> 5);
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
87         ↪ unchecked(storedValue & 31UL | (size & 134217727UL) << 5);
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     protected override bool GetLeftIsChildValue(ulong value) => unchecked((value & 16UL) >>
90         ↪ 4 == 1UL);
91     [MethodImpl(MethodImplOptions.AggressiveInlining)]
92     protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
93         ↪ storedValue = unchecked(storedValue & 4294967279UL | (As<bool, byte>(ref value) &
94         ↪ 1UL) << 4);
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     protected override bool GetRightIsChildValue(ulong value) => unchecked((value & 8UL) >>
97         ↪ 3 == 1UL);
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>
100         ↪ storedValue = unchecked(storedValue & 4294967287UL | (As<bool, byte>(ref value) &
101         ↪ 1UL) << 3);
102     [MethodImpl(MethodImplOptions.AggressiveInlining)]
103     protected override sbyte GetBalanceValue(ulong value) => unchecked((sbyte)(value & 7UL |
104         ↪ 0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of
105         ↪ sbyte
106     [MethodImpl(MethodImplOptions.AggressiveInlining)]
107     protected override void SetBalanceValue(ref ulong storedValue, sbyte value) =>
108         ↪ storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
109         ↪ value & 3) & 7UL);
110     [MethodImpl(MethodImplOptions.AggressiveInlining)]
111     protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
112     [MethodImpl(MethodImplOptions.AggressiveInlining)]
113     protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
114 }

```

./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 IsEquals(ulong first, ulong second) => first == second;

```

```

29 [MethodImpl(MethodImplOptions.AggressiveInlining)]
30 protected override bool GreaterThanZero(ulong value) => value > 0UL;
31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 protected override bool GreaterThan(ulong first, ulong second) => first > second;
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
37
38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
    ↳ always true for ulong
40
41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
    ↳ always >= 0 for ulong
43
44 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
    ↳ for ulong
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 protected override bool LessThan(ulong first, ulong second) => first < second;
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected override ulong Increment(ulong value) => ++value;
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 protected override ulong Decrement(ulong value) => --value;
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected override ulong Add(ulong first, ulong second) => first + second;
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override ulong Subtract(ulong first, ulong second) => first - second;
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
67 {
68     ref var firstLink = ref Links[first];
69     ref var secondLink = ref Links[second];
70     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
    ↳ secondLink.Source, secondLink.Target);
71 }
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
75 {
76     ref var firstLink = ref Links[first];
77     ref var secondLink = ref Links[second];
78     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↳ secondLink.Source, secondLink.Target);
79 }
80
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
86 }
87 }

```

./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 :
    ↳ UInt64LinksAvlBalancedTreeMethodsBase
8     {
9         public UInt64LinksSourcesAvlBalancedTreeMethods(LinksConstants<ulong> constants,
    ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
    ↳ { }
10

```

```

11 [MethodImpl(MethodImplOptions.AggressiveInlining)]
12 protected override ref ulong GetLeftReference(ulong node) => ref
    ↳ Links[node].LeftAsSource;
13
14 [MethodImpl(MethodImplOptions.AggressiveInlining)]
15 protected override ref ulong GetRightReference(ulong node) => ref
    ↳ Links[node].RightAsSource;
16
17 [MethodImpl(MethodImplOptions.AggressiveInlining)]
18 protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
19
20 [MethodImpl(MethodImplOptions.AggressiveInlining)]
21 protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
22
23 [MethodImpl(MethodImplOptions.AggressiveInlining)]
24 protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
    ↳ left;
25
26 [MethodImpl(MethodImplOptions.AggressiveInlining)]
27 protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
    ↳ right;
28
29 [MethodImpl(MethodImplOptions.AggressiveInlining)]
30 protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsSource);
31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
    ↳ Links[node].SizeAsSource, size);
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 protected override bool GetLeftIsChild(ulong node) =>
    ↳ GetLeftIsChildValue(Links[node].SizeAsSource);
37
38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 protected override void SetLeftIsChild(ulong node, bool value) =>
    ↳ SetLeftIsChildValue(ref Links[node].SizeAsSource, value);
40
41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 protected override bool GetRightIsChild(ulong node) =>
    ↳ GetRightIsChildValue(Links[node].SizeAsSource);
43
44 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 protected override void SetRightIsChild(ulong node, bool value) =>
    ↳ SetRightIsChildValue(ref Links[node].SizeAsSource, value);
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 protected override sbyte GetBalance(ulong node) =>
    ↳ GetBalanceValue(Links[node].SizeAsSource);
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
    ↳ Links[node].SizeAsSource, value);
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected override ulong GetTreeRoot() => Header->FirstAsSource;
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
    ↳ ulong secondSource, ulong secondTarget)
61     => firstSource < secondSource || firstSource == secondSource && firstTarget <
    ↳ secondTarget;
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
    ↳ ulong secondSource, ulong secondTarget)
65     => firstSource > secondSource || firstSource == secondSource && firstTarget >
    ↳ secondTarget;
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 protected override void ClearNode(ulong node)
69 {
70     ref var link = ref Links[node];
71     link.LeftAsSource = OUL;
72     link.RightAsSource = OUL;
73     link.SizeAsSource = OUL;

```

```

74     }
75 }
76 }

```

./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
11         public UInt64LinksSourcesSizeBalancedTreeMethods(LinksConstants<ulong> constants,
12             ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
13             ↳ { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected override ref ulong GetLeftReference(ulong node) => ref
17             ↳ Links[node].LeftAsSource;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected override ref ulong GetRightReference(ulong node) => ref
21             ↳ Links[node].RightAsSource;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
31             ↳ left;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
35             ↳ right;
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected override ulong GetSize(ulong node) => Links[node].SizeAsSource;
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsSource =
42             ↳ size;
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override ulong GetTreeRoot() => Header->FirstAsSource;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
52             ↳ ulong secondSource, ulong secondTarget)
53             ↳ => firstSource < secondSource || firstSource == secondSource && firstTarget <
54                 ↳ secondTarget;
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
58             ↳ ulong secondSource, ulong secondTarget)
59             ↳ => firstSource > secondSource || firstSource == secondSource && firstTarget >
60                 ↳ secondTarget;
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         protected override void ClearNode(ulong node)
64         {
65             ref var link = ref Links[node];
66             link.LeftAsSource = OUL;
67             link.RightAsSource = OUL;
68             link.SizeAsSource = OUL;
69         }
70     }
71 }

```

./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) => GetSizeValue(Links[node].SizeAsTarget);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
41         ↪ Links[node].SizeAsTarget, size);
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override bool GetLeftIsChild(ulong node) =>
45         ↪ GetLeftIsChildValue(Links[node].SizeAsTarget);
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override void SetLeftIsChild(ulong node, bool value) =>
49         ↪ SetLeftIsChildValue(ref Links[node].SizeAsTarget, value);
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override bool GetRightIsChild(ulong node) =>
53         ↪ GetRightIsChildValue(Links[node].SizeAsTarget);
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override void SetRightIsChild(ulong node, bool value) =>
57         ↪ SetRightIsChildValue(ref Links[node].SizeAsTarget, value);
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override sbyte GetBalance(ulong node) =>
61         ↪ GetBalanceValue(Links[node].SizeAsTarget);
62
63         [MethodImpl(MethodImplOptions.AggressiveInlining)]
64         protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
65         ↪ Links[node].SizeAsTarget, value);
66
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         protected override ulong GetTreeRoot() => Header->FirstAsTarget;
69
70         [MethodImpl(MethodImplOptions.AggressiveInlining)]
71         protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
72
73         [MethodImpl(MethodImplOptions.AggressiveInlining)]
74         protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
75         ↪ ulong secondSource, ulong secondTarget)
76         ↪ => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
77         ↪ secondSource;
78
79         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

64     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
65         ↪     ulong secondSource, ulong secondTarget)
66         => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
67         ↪     secondSource;
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override void ClearNode(ulong node)
71     {
72         ref var link = ref Links[node];
73         link.LeftAsTarget = OUL;
74         link.RightAsTarget = OUL;
75         link.SizeAsTarget = OUL;
76     }
77 }

```

./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 :
8          ↪     UInt64LinksSizeBalancedTreeMethodsBase
9      {
10         public UInt64LinksTargetsSizeBalancedTreeMethods(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             => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
53             ↪     secondSource;
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
57             ↪     ulong secondSource, ulong secondTarget)
58             => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
59             ↪     secondSource;
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         protected override void ClearNode(ulong node)

```

```

51     {
52         ref var link = ref Links[node];
53         link.LeftAsTarget = OUL;
54         link.RightAsTarget = OUL;
55         link.SizeAsTarget = OUL;
56     }
57 }
58 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Memory;
5  using Platform.Data.Doublets.ResizableDirectMemory.Generic;
6  using Platform.Singletons;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
11 {
12     public unsafe class UInt64ResizableDirectMemoryLinks : ResizableDirectMemoryLinksBase<ulong>
13     {
14         private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
15         private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
16         private LinksHeader<ulong>* _header;
17         private RawLink<ulong>* _links;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UInt64ResizableDirectMemoryLinks(string address) : this(address,
21             ↪ DefaultLinksSizeStep) { }
22
23         /// <summary>
24         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
25         ↪ минимальным шагом расширения базы данных.
26         /// </summary>
27         /// <param name="address">Полный путь к файлу базы данных.</param>
28         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
29         ↪ байтах.</param>
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
32             ↪ this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
33             ↪ memoryReservationStep) { }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
37             ↪ DefaultLinksSizeStep) { }
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
41             ↪ memoryReservationStep) : this(memory, memoryReservationStep,
42             ↪ Default<LinksConstants<ulong>>.Instance, true) { }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
46             ↪ memoryReservationStep, LinksConstants<ulong> constants, bool useAvlBasedIndex) :
47             ↪ base(memory, memoryReservationStep, constants)
48         {
49             if (useAvlBasedIndex)
50             {
51                 _createSourceTreeMethods = () => new
52                 ↪ UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
53                 _createTargetTreeMethods = () => new
54                 ↪ UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
55             }
56             else
57             {
58                 _createSourceTreeMethods = () => new
59                 ↪ UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
60                 _createTargetTreeMethods = () => new
61                 ↪ UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
62             }
63             Init(memory, memoryReservationStep);
64         }
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         protected override void SetPointers(IResizableDirectMemory memory)
68         {
69             _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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 protected override IList<ulong> GetEmptyList() => new ulong[0];
117 }
118 }

```

./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 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences
7  {
8      public static class ArrayExtensions
9      {
10         public static IList<TLink> ConvertToRestrictionsValues<TLink>(this TLink[] array)
11         {
12             var restrictions = new TLink[array.Length + 1];
13             Array.Copy(array, 0, restrictions, 1, array.Length);
14             return restrictions;
15         }
16     }
17 }

```

./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(IList<TLink> sequence)
12         {
13             var length = sequence.Count;
14             if (length < 1)
15             {
16                 return default;
17             }
18             if (length == 1)
19             {
20                 return sequence[0];
21             }
22             // Make copy of next layer
23             if (length > 2)
24             {
25                 // TODO: Try to use stackalloc (which at the moment is not working with
26                 // ↪ generics) but will be possible with Sigil
27                 var halvedSequence = new TLink[(length / 2) + (length % 2)];
28                 HalveSequence(halvedSequence, sequence, length);
29                 sequence = halvedSequence;
30                 length = halvedSequence.Length;
31             }
32             // Keep creating layer after layer
33             while (length > 2)
34             {
35                 HalveSequence(sequence, sequence, length);
36                 length = (length / 2) + (length % 2);
37             }
38             return Links.GetOrCreate(sequence[0], sequence[1]);
39         }
40
41         private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
42         {
43             var loopedLength = length - (length % 2);
44             for (var i = 0; i < loopedLength; i += 2)
45             {

```

```

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 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5  using Platform.Collections;
6  using Platform.Singletons;
7  using Platform.Numbers;
8  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.Sequences.Converters
13 {
14     /// <remarks>
15     /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от
16     ///     Links на этапе сжатия.
17     ///     А именно будет создаваться временный список пар необходимых для выполнения сжатия, в
18     ///     таком случае тип значения элемента массива может быть любым, как char так и ulong.
19     ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
20     ///     пар, а так же разом выполнить замену.
21     /// </remarks>
22     public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
23     {
24         private static readonly LinksConstants<TLink> _constants =
25             ↳ Default<LinksConstants<TLink>>.Instance;
26         private static readonly EqualityComparer<TLink> _equalityComparer =
27             ↳ EqualityComparer<TLink>.Default;
28         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
29
30         private readonly IConverter<IList<TLink>, TLink> _baseConverter;
31         private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
32         private readonly TLink _minFrequencyToCompress;
33         private readonly bool _doInitialFrequenciesIncrement;
34         private Doublet<TLink> _maxDoublet;
35         private LinkFrequency<TLink> _maxDoubletData;
36
37         private struct HalfDoublet
38         {
39             public TLink Element;
40             public LinkFrequency<TLink> DoubletData;
41
42             public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
43             {
44                 Element = element;
45                 DoubletData = doubletData;
46             }
47
48             public override string ToString() => $"{Element}: ({DoubletData})";
49         }
50
51         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
52             ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
53             : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One, true)
54         {
55         }
56
57         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
58             ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
59             ↳ doInitialFrequenciesIncrement)
60             : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One,
61                 ↳ doInitialFrequenciesIncrement)
62         {
63         }
64
65         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
66             ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
67             ↳ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
68             : base(links)
69         {
70         }
71     }
72 }

```

```

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(ICollection<TLink> source) =>
71     ↪ _baseConverter.Convert(Compress(source));
72
73 /// <remarks>
74 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
75 /// Faster version (doublets' frequencies dictionary is not recreated).
76 /// </remarks>
77 private ICollection<TLink> Compress(ICollection<TLink> sequence)
78 {
79     if (sequence.IsNullOrEmpty())
80     {
81         return null;
82     }
83     if (sequence.Count == 1)
84     {
85         return sequence;
86     }
87     if (sequence.Count == 2)
88     {
89         return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
90     }
91     // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
92     var copy = new HalfDoublet[sequence.Count];
93     Doublet<TLink> doublet = default;
94     for (var i = 1; i < sequence.Count; i++)
95     {
96         doublet.Source = sequence[i - 1];
97         doublet.Target = sequence[i];
98         LinkFrequency<TLink> data;
99         if (_doInitialFrequenciesIncrement)
100         {
101             data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
102         }
103         else
104         {
105             data = _doubletFrequenciesCache.GetFrequency(ref doublet);
106             if (data == null)
107             {
108                 throw new NotSupportedException("If you ask not to increment
109                 ↪ frequencies, it is expected that all frequencies for the sequence
110                 ↪ are prepared.");
111             }
112         }
113         copy[i - 1].Element = sequence[i - 1];
114         copy[i - 1].DoubletData = data;
115         UpdateMaxDoublet(ref doublet, data);
116     }
117     copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
118     copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
119     if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
120     {
121         var newLength = ReplaceDoublets(copy);
122         sequence = new TLink[newLength];
123         for (int i = 0; i < newLength; i++)
124         {
125             sequence[i] = copy[i].Element;
126         }
127     }
128     return sequence;
129 }
130
131 /// <remarks>
132 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
133 /// </remarks>
134 private int ReplaceDoublets(HalfDoublet[] copy)
135 {
136     var oldLength = copy.Length;
137     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) &&
151             ⇨ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
152         {
153             if (r > 0)
154             {
155                 var previous = copy[w - 1].Element;
156                 copy[w - 1].DoubletData.DecrementFrequency();
157                 copy[w - 1].DoubletData =
158                     ⇨ _doubletFrequenciesCache.IncrementFrequency(previous,
159                     ⇨ maxDoubletReplacementLink);
160             }
161             if (r < oldLengthMinusTwo)
162             {
163                 var next = copy[r + 2].Element;
164                 copy[r + 1].DoubletData.DecrementFrequency();
165                 copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
166                     ⇨ next);
167             }
168             copy[w++].Element = maxDoubletReplacementLink;
169             r++;
170             newLength--;
171         }
172         else
173         {
174             copy[w++] = copy[r];
175         }
176     }
177     if (w < newLength)
178     {
179         copy[w] = copy[r];
180     }
181     oldLength = newLength;
182     ResetMaxDoublet();
183     UpdateMaxDoublet(copy, newLength);
184 }
185 return newLength;
186 }
187
188 [MethodImpl(MethodImplOptions.AggressiveInlining)]
189 private void ResetMaxDoublet()
190 {
191     _maxDoublet = new Doublet<TLink>();
192     _maxDoubletData = new LinkFrequency<TLink>();
193 }
194
195 [MethodImpl(MethodImplOptions.AggressiveInlining)]
196 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
197 {
198     Doublet<TLink> doublet = default;
199     for (var i = 1; i < length; i++)
200     {
201         doublet.Source = copy[i - 1].Element;
202         doublet.Target = copy[i].Element;
203         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
204     }
205 }
206
207 [MethodImpl(MethodImplOptions.AggressiveInlining)]
208 private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
209 {
210     var frequency = data.Frequency;
211     var maxFrequency = _maxDoubletData.Frequency;

```

```

208 //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
    ↳ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
    ↳ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
    ↳ _maxDoublet.Target)))
209 if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
210     (_comparer.Compare(maxFrequency, frequency) < 0 ||
    ↳ (_equalityComparer.Equals(maxFrequency, frequency) &&
    ↳ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
    ↳ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
    ↳ better stability and better compression on sequent data and even on random
    ↳ numbers data (but gives collisions anyway) */
211 {
212     _maxDoublet = doublet;
213     _maxDoubletData = data;
214 }
215 }
216 }
217 }

```

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

```

./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.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.Converters
8 {
9     public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↳ EqualityComparer<TLink>.Default;
12         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
13
14         private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
15
16         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
    ↳ sequenceToItsLocalElementLevelsConverter) : base(links)
17         => _sequenceToItsLocalElementLevelsConverter =
    ↳ sequenceToItsLocalElementLevelsConverter;
18
19         public override TLink Convert(IList<TLink> sequence)
20         {
21             var length = sequence.Count;
22             if (length == 1)
23             {
24                 return sequence[0];
25             }
26             var links = Links;
27             if (length == 2)
28             {
29                 return links.GetOrCreate(sequence[0], sequence[1]);
30             }
31             sequence = sequence.ToArray();
32             var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
33             while (length > 2)
34             {
35                 var levelRepeat = 1;
36                 var currentLevel = levels[0];
37                 var previousLevel = levels[0];
38                 var skipOnce = false;
39                 var w = 0;

```

```

40     for (var i = 1; i < length; i++)
41     {
42         if (_equalityComparer.Equals(currentLevel, levels[i]))
43         {
44             levelRepeat++;
45             skipOnce = false;
46             if (levelRepeat == 2)
47             {
48                 sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
49                 var newLevel = i >= length - 1 ?
50                     GetPreviousLowerThanCurrentOrCurrent(previousLevel,
51                     ↪ currentLevel) :
52                     i < 2 ?
53                     GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
54                     GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
55                     ↪ currentLevel, levels[i + 1]);
56                 levels[w] = newLevel;
57                 previousLevel = currentLevel;
58                 w++;
59                 levelRepeat = 0;
60                 skipOnce = true;
61             }
62             else if (i == length - 1)
63             {
64                 sequence[w] = sequence[i];
65                 levels[w] = levels[i];
66                 w++;
67             }
68         }
69         else
70         {
71             currentLevel = levels[i];
72             levelRepeat = 1;
73             if (skipOnce)
74             {
75                 skipOnce = false;
76             }
77             else
78             {
79                 sequence[w] = sequence[i - 1];
80                 levels[w] = levels[i - 1];
81                 previousLevel = levels[w];
82                 w++;
83             }
84             if (i == length - 1)
85             {
86                 sequence[w] = sequence[i];
87                 levels[w] = levels[i];
88                 w++;
89             }
90         }
91     }
92     length = w;
93     return links.GetOrCreate(sequence[0], sequence[1]);
94 }
95 private static TLink GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink previous, TLink
96 ↪ current, TLink next)
97 {
98     return _comparer.Compare(previous, next) > 0
99         ? _comparer.Compare(previous, current) < 0 ? previous : current
100         : _comparer.Compare(next, current) < 0 ? next : current;
101 }
102 private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
103     ↪ _comparer.Compare(next, current) < 0 ? next : current;
104 private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
105     ↪ => _comparer.Compare(previous, current) < 0 ? previous : current;
106 }

```

./Platform.Data.Doublets/Sequences/Converters/SequenceToItsLocalElementLevelsConverter.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.Converters
7 {
8     public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
        ↳ IConverter<IList<TLink>>
9     {
10         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
11         private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
12
13         public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
        ↳ IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
        ↳ => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
14
15         public IList<TLink> Convert(IList<TLink> sequence)
16         {
17             var levels = new TLink[sequence.Count];
18             levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
19             for (var i = 1; i < sequence.Count - 1; i++)
20             {
21                 var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
22                 var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
23                 levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
24             }
25             levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
        ↳ sequence[sequence.Count - 1]);
26             return levels;
27         }
28     }
29
30     public TLink GetFrequencyNumber(TLink source, TLink target) =>
        ↳ _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
31 }
32 }

```

./Platform.Data.Doublets/Sequences/CreteriaMatchers/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.CreteriaMatchers
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 }

```

./Platform.Data.Doublets/Sequences/CreteriaMatchers/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.CreteriaMatchers
7 {
8     public class MarkedSequenceCriterionMatcher<TLink> : ICriterionMatcher<TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
        ↳ EqualityComparer<TLink>.Default;
11
12         private readonly ILinks<TLink> _links;
13         private readonly TLink _sequenceMarkerLink;
14
15         public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
16         {
17             _links = links;
18             _sequenceMarkerLink = sequenceMarkerLink;
19         }
20
21         public bool IsMatched(TLink sequenceCandidate)
22             => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
23             || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
        ↳ sequenceCandidate), _links.Constants.Null);
24     }
25 }

```


./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
27         public TLink Append(TLink sequence, TLink appendant)
28         {
29             var cursor = sequence;
30             while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
31             {
32                 var source = Links.GetSource(cursor);
33                 var target = Links.GetTarget(cursor);
34                 if (_equalityComparer.Equals(_heightProvider.Get(source),
35                     ↳ _heightProvider.Get(target)))
36                 {
37                     break;
38                 }
39                 else
40                 {
41                     _stack.Push(source);
42                     cursor = target;
43                 }
44             }
45             var left = cursor;
46             var right = appendant;
47             while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
48             {
49                 right = Links.GetOrCreate(left, right);
50                 left = cursor;
51             }
52             return Links.GetOrCreate(left, right);
53         }
54     }
55 }
```

./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 }
```

./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                 return intermediateResult;
56             }
57         }
58
59         public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
60             : base(minimumStringSegmentLength: 2)
61         {
62             _links = links;
63             _sequences = sequences;
64         }
65
66         public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
67         {
68             _groups = new HashSet<KeyValuePair<IList<TLink>,
69                 ↳ IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
70             var count = _links.Count();
71             _visited = new BitString((long)(Integer<TLink>)count + 1);
72             _links.Each(link =>
73             {
74                 var linkIndex = _links.GetIndex(link);
75                 var linkBitIndex = (long)(Integer<TLink>)linkIndex;
76                 if (!_visited.Get(linkBitIndex))
77                 {
78                     var sequenceElements = new List<TLink>();
79                     var filler = new ListFiller<TLink, TLink>(sequenceElements,
80                         ↳ _sequences.Constants.Break);
81                     _sequences.Each(filler.AddAllValuesAndReturnConstant, new
82                         ↳ LinkAddress<TLink>(linkIndex));
93                 }
94             });
95         }
96     }
97 }

```

```

70         if (sequenceElements.Count > 2)
71         {
72             WalkAll(sequenceElements);
73         }
74     }
75     return _links.Constants.Continue;
76 });
77 var resultList = _groups.ToList();
78 var comparer = Default<ItemComparer>.Instance;
79 resultList.Sort(comparer);
80 #if DEBUG
81     foreach (var item in resultList)
82     {
83         PrintDuplicates(item);
84     }
85 #endif
86     return resultList;
87 }
88
89 protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
    ↪ length) => new Segment<TLink>(elements, offset, length);
90
91 protected override void OnDuplicateFound(Segment<TLink> segment)
92 {
93     var duplicates = CollectDuplicatesForSegment(segment);
94     if (duplicates.Count > 1)
95     {
96         _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
    ↪ duplicates));
97     }
98 }
99
100 private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
101 {
102     var duplicates = new List<TLink>();
103     var readAsElement = new HashSet<TLink>();
104     var restrictions = segment.ConvertToRestrictionsValues();
105     restrictions[0] = _sequences.Constants.Any;
106     _sequences.Each(sequence =>
107     {
108         var sequenceIndex = sequence[_sequences.Constants.IndexPart];
109         duplicates.Add(sequenceIndex);
110         readAsElement.Add(sequenceIndex);
111         return _sequences.Constants.Continue;
112     }, restrictions);
113     if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
114     {
115         return new List<TLink>();
116     }
117     foreach (var duplicate in duplicates)
118     {
119         var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
120         _visited.Set(duplicateBitIndex);
121     }
122     if (_sequences is Sequences sequencesExperiments)
123     {
124         var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
    ↪ ashSet<ulong>)(object)readAsElement,
    ↪ (IList<ulong>)segment);
125         foreach (var partiallyMatchedSequence in partiallyMatched)
126         {
127             TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
128             duplicates.Add(sequenceIndex);
129         }
130     }
131     duplicates.Sort();
132     return duplicates;
133 }
134
135 private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
136 {
137     if (!(_links is ILinks<ulong> ulongLinks))
138     {
139         return;
140     }
141     var duplicatesKey = duplicatesItem.Key;
142     var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
143     Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)}");

```

```

144     var duplicatesList = duplicatesItem.Value;
145     for (int i = 0; i < duplicatesList.Count; i++)
146     {
147         ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
148         var formattedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
            ↳ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
            ↳ UnicodeMap.IsCharLink(link.Index) ?
            ↳ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
149         Console.WriteLine(formattedSequenceStructure);
150         var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
            ↳ ulongLinks);
151         Console.WriteLine(sequenceString);
152     }
153     Console.WriteLine();
154 }
155 }
156 }

```

./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)]
34         public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
35         {
36             var doublet = new Doublet<TLink>(source, target);
37             return GetFrequency(ref doublet);
38         }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
42         {
43             _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
44             return data;
45         }
46
47         public void IncrementFrequencies(IList<TLink> sequence)
48         {
49             for (var i = 1; i < sequence.Count; i++)
50             {
51                 IncrementFrequency(sequence[i - 1], sequence[i]);
52             }
53         }
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
57         {
58             var doublet = new Doublet<TLink>(source, target);
59             return IncrementFrequency(ref doublet);
60         }
61     }
62 }

```

```

58
59 public void PrintFrequencies(IList<TLink> sequence)
60 {
61     for (var i = 1; i < sequence.Count; i++)
62     {
63         PrintFrequency(sequence[i - 1], sequence[i]);
64     }
65 }
66
67 public void PrintFrequency(TLink source, TLink target)
68 {
69     var number = GetFrequency(source, target).Frequency;
70     Console.WriteLine("{0},{1} - {2}", source, target, number);
71 }
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
75 {
76     if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
77     {
78         data.IncrementFrequency();
79     }
80     else
81     {
82         var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
83         data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
84         if (!_equalityComparer.Equals(link, default))
85         {
86             data.Frequency = Arithmetic.Add(data.Frequency,
87                 ↪ _frequencyCounter.Count(link));
88         }
89         _doubletsCache.Add(doublet, data);
90     }
91     return data;
92 }
93
94 public void ValidateFrequencies()
95 {
96     foreach (var entry in _doubletsCache)
97     {
98         var value = entry.Value;
99         var linkIndex = value.Link;
100         if (!_equalityComparer.Equals(linkIndex, default))
101         {
102             var frequency = value.Frequency;
103             var count = _frequencyCounter.Count(linkIndex);
104             // TODO: Why `frequency` always greater than `count` by 1?
105             if (((_comparer.Compare(frequency, count) > 0) &&
106                 ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count),
107                 ↪ Integer<TLink>.One) > 0))
108                 || ((_comparer.Compare(count, frequency) > 0) &&
109                 ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency),
110                 ↪ Integer<TLink>.One) > 0)))
111             {
112                 throw new InvalidOperationException("Frequencies validation failed.");
113             }
114             //else
115             //{
116             //    if (value.Frequency > 0)
117             //    {
118             //        var frequency = value.Frequency;
119             //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
120             //        var count = _countLinkFrequency(linkIndex);
121             //        if ((frequency > count && frequency - count > 1) || (count > frequency
122             //            ↪ && count - frequency > 1))
123             //            throw new Exception("Frequencies validation failed.");
124             //    }
125             //}
126         }
127     }
128 }

```

./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 }

```

./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToItsFrequencyValueConverter.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.Cache
6 {
7     public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
8         ⇨ IConverter<Doublet<TLink>, TLink>
9     {
10         private readonly LinkFrequenciesCache<TLink> _cache;
11         public
12             ⇨ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
13             ⇨ cache) => _cache = cache;
14         public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
15     }
16 }

```

./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 }

```

./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 }

```

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

```

./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 }

```

./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 }

```

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

```

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         {

```



```

39     var any = _links.Constants.Any;
40     if (_equalityComparer.Equals(_links.Count(any, link), default))
41     {
42         CountSequenceSymbolFrequency(link);
43     }
44     else
45     {
46         _links.Each(EachElementHandler, any, link);
47     }
48 }
49
50 protected virtual void CountSequenceSymbolFrequency(TLink link)
51 {
52     var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
53     ↪ link, _symbol);
54     _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
55 }
56
57 private TLink EachElementHandler(IList<TLink> doublet)
58 {
59     var constants = _links.Constants;
60     var doubletIndex = doublet[constants.IndexPart];
61     if (_visits.Add(doubletIndex))
62     {
63         CountCore(doubletIndex);
64     }
65     return constants.Continue;
66 }
67 }

```

./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.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.HeightProviders
7  {
8      public class CachedSequenceHeightProvider<TLink> : LinksOperatorBase<TLink>,
9      ↪ ISequenceHeightProvider<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12         ↪ EqualityComparer<TLink>.Default;
13
14         private readonly TLink _heightPropertyMarker;
15         private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
16         private readonly IConverter<TLink> _addressToUnaryNumberConverter;
17         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
18         private readonly IPropertiesOperator<TLink, TLink, TLink> _propertyOperator;
19
20         public CachedSequenceHeightProvider(
21             ILinks<TLink> links,
22             ISequenceHeightProvider<TLink> baseHeightProvider,
23             IConverter<TLink> addressToUnaryNumberConverter,
24             IConverter<TLink> unaryNumberToAddressConverter,
25             TLink heightPropertyMarker,
26             IPropertiesOperator<TLink, TLink, TLink> propertyOperator)
27             : base(links)
28         {
29             _heightPropertyMarker = heightPropertyMarker;
30             _baseHeightProvider = baseHeightProvider;
31             _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
32             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
33             _propertyOperator = propertyOperator;
34         }
35
36         public TLink Get(TLink sequence)
37         {
38             TLink height;
39             var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
40             if (_equalityComparer.Equals(heightValue, default))
41             {
42                 height = _baseHeightProvider.Get(sequence);
43                 heightValue = _addressToUnaryNumberConverter.Convert(height);
44                 _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
45             }
46             else
47             {
48                 height = _unaryNumberToAddressConverter.Convert(heightValue);
49             }
50         }
51     }
52 }

```

```

47     }
48     return height;
49 }
50 }
51 }

```

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

```

1  using Platform.Interfaces;
2  using Platform.Numbers;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.HeightProviders
7  {
8      public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
9          ↳ ISequenceHeightProvider<TLink>
10     {
11         private readonly ICriterionMatcher<TLink> _elementMatcher;
12
13         public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
14             ↳ elementMatcher) : base(links) => _elementMatcher = elementMatcher;
15
16         public TLink Get(TLink sequence)
17         {
18             var height = default(TLink);
19             var pairOrElement = sequence;
20             while (!_elementMatcher.IsMatched(pairOrElement))
21             {
22                 pairOrElement = Links.GetTarget(pairOrElement);
23                 height = Arithmetic.Increment(height);
24             }
25             return height;
26         }
27     }
28 }

```

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

```

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

```

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

```

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

```

```

31         return restrictions;
32     }
33 }
34 }

```

./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(ICollection<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(ICollection<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 }

```

./Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.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.Sequences.Indexes
7  {

```

```

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

```

./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 }

```

./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 =
    ↪ EqualityComparer<TLink>.Default;
10
11     public SequenceIndex(ILinks<TLink> links) : base(links) { }
12
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 }

```

./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 }

```

./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 }

```

./Platform.Data.Doublets/Sequences/ListFiller.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences
7 {
8     public class ListFiller<TElement, TReturnConstant>
9     {
10         protected readonly List<TElement> _list;
11         protected readonly TReturnConstant _returnConstant;
12
13         public ListFiller(List<TElement> list, TReturnConstant returnConstant)
14         {
15             _list = list;
16             _returnConstant = returnConstant;
17         }
18
19         public ListFiller(List<TElement> list) : this(list, default) { }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public void Add(TElement element) => _list.Add(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public bool AddAndReturnTrue(TElement element)
26         {
27             _list.Add(element);
28             return true;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public bool AddFirstAndReturnTrue(IList<TElement> collection)
33         {
34             _list.Add(collection[0]);
35             return true;
36         }
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public TReturnConstant AddAndReturnConstant(TElement element)
40         {
41             _list.Add(element);
42             return _returnConstant;
43         }
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public TReturnConstant AddFirstAndReturnConstant(IList<TElement> collection)
47         {
48             _list.Add(collection[0]);
49             return _returnConstant;
50         }
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         public TReturnConstant AddAllValuesAndReturnConstant(IList<TElement> collection)
54         {
55             for (int i = 1; i < collection.Count; i++)

```

```

56         {
57             _list.Add(collection[i]);
58         }
59         return _returnConstant;
60     }
61 }
62 }

```

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

```

1  using Platform.Collections;
2  using Platform.Collections.Lists;
3  using Platform.Collections.Stacks;
4  using Platform.Data.Doublets.Sequences.Walkers;
5  using Platform.Singletons;
6  using Platform.Threading.Synchronization;
7  using System;
8  using System.Collections.Generic;
9  using System.Linq;
10 using System.Runtime.CompilerServices;
11 using LinkIndex = System.UInt64;
12
13 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14
15 namespace Platform.Data.Doublets.Sequences
16 {
17     /// <summary>
18     /// Представляет коллекцию последовательностей связей.
19     /// </summary>
20     /// <remarks>
21     /// Обязательно реализовать атомарность каждого публичного метода.
22     ///
23     /// TODO:
24     ///
25     /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
26     /// через естественную группировку по unicode типам, все whitespace вместе, все символы
27     /// ↪ вместе, все числа вместе и т.п.
28     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
29     /// ↪ графа)
30     ///
31     /// х*у - найти все связи между, в последовательностях любой формы, если не стоит
32     /// ↪ ограничитель на то, что является последовательностью, а что нет,
33     /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
34     /// ↪ порядке.
35     ///
36     /// Рост последовательности слева и справа.
37     /// Поиск со звёздочкой.
38     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
39     /// так же проблема может быть решена при реализации дистанционных триггеров.
40     /// Нужны ли уникальные указатели вообще?
41     /// Что если обращение к информации будет происходить через содержимое всегда?
42     ///
43     /// Писать тесты.
44     ///
45     /// Можно убрать зависимость от конкретной реализации Links,
46     /// на зависимость от абстрактного элемента, который может быть представлен несколькими
47     /// ↪ способами.
48     ///
49     /// Можно ли как-то сделать один общий интерфейс
50     ///
51     /// Блокчейн и/или гит для распределённой записи транзакций.
52     ///
53     /// </remarks>
54     public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
55     ↪ (после завершения реализации Sequences)
56     {
57         /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
58         /// ↪ связей.</summary>
59         public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
60
61         public SequencesOptions<LinkIndex> Options { get; }
62         public SynchronizedLinks<LinkIndex> Links { get; }
63         private readonly ISynchronization _sync;
64
65         public LinksConstants<LinkIndex> Constants { get; }
66
67         public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
68         {

```

```

64     Links = links;
65     _sync = links.SyncRoot;
66     Options = options;
67     Options.ValidateOptions();
68     Options.InitOptions(Links);
69     Constants = links.Constants;
70 }
71
72 public Sequences(SynchronizedLinks<LinkIndex> links)
73     : this(links, new SequencesOptions<LinkIndex>())
74 {
75 }
76
77 public bool IsSequence(LinkIndex sequence)
78 {
79     return _sync.ExecuteReadOperation(() =>
80     {
81         if (Options.UseSequenceMarker)
82         {
83             return Options.MarkedSequenceMatcher.IsMatched(sequence);
84         }
85         return !Links.Unsync.IsPartialPoint(sequence);
86     });
87 }
88
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 private LinkIndex GetSequenceByElements(LinkIndex sequence)
91 {
92     if (Options.UseSequenceMarker)
93     {
94         return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
95     }
96     return sequence;
97 }
98
99 private LinkIndex GetSequenceElements(LinkIndex sequence)
100 {
101     if (Options.UseSequenceMarker)
102     {
103         var linkContents = new Link<ulong>(Links.GetLink(sequence));
104         if (linkContents.Source == Options.SequenceMarkerLink)
105         {
106             return linkContents.Target;
107         }
108         if (linkContents.Target == Options.SequenceMarkerLink)
109         {
110             return linkContents.Source;
111         }
112     }
113     return sequence;
114 }
115
116 #region Count
117
118 public LinkIndex Count(IList<LinkIndex> restrictions)
119 {
120     if (restrictions.IsNullOrEmpty())
121     {
122         return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
123     }
124     if (restrictions.Count == 1) // Первая связь это адрес
125     {
126         var sequenceIndex = restrictions[0];
127         if (sequenceIndex == Constants.Null)
128         {
129             return 0;
130         }
131         if (sequenceIndex == Constants.Any)
132         {
133             return Count(null);
134         }
135         if (Options.UseSequenceMarker)
136         {
137             return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
138         }
139         return Links.Exists(sequenceIndex) ? 1UL : 0;
140     }
141     throw new NotImplementedException();
142 }

```



```

143
144 private LinkIndex CountUsages(params LinkIndex[] restrictions)
145 {
146     if (restrictions.Length == 0)
147     {
148         return 0;
149     }
150     if (restrictions.Length == 1) // Первая связь это адрес
151     {
152         if (restrictions[0] == Constants.Null)
153         {
154             return 0;
155         }
156         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(sequenceLink) + Links.Count(elementsLink) - 1;
163             }
164             return Links.Count(elementsLink);
165         }
166         return Links.Count(restrictions[0]);
167     }
168     throw new NotImplementedException();
169 }
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.ExtractValues();
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         Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
215     }
216     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
217 }
218
219 #endregion
220
221 #region Each

```

```

222 public List<LinkIndex> Each(IList<LinkIndex> sequence)
223 {
224     var results = new List<LinkIndex>();
225     var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
226     Each(filler.AddFirstAndReturnConstant, sequence);
227     return results;
228 }
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().ConvertToRestrictionsValues();
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.ExtractValues();
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         return Constants.Break;
302     }
303 }
304 if (values.Count >= 3)
305 {
306     if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
        ↳ != Constants.Continue)
        {
307         return Constants.Break;
308     }
309 }
310 }
311 return Constants.Continue;
312 }
313
314 private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ left, LinkIndex right)
315 {
316     return Links.Unsync.Each(doublet =>
317     {
318         var doubletIndex = doublet[Constants.IndexPart];
319         if (StepRight(handler, doubletIndex, right) != Constants.Continue)
320         {
321             return Constants.Break;
322         }
323         if (left != doubletIndex)
324         {
325             return PartialStepRight(handler, doubletIndex, right);
326         }
327         return Constants.Continue;
328     }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
329 }
330
331 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));
332
333 private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ right, LinkIndex stepFrom)
334 {
335     var upStep = stepFrom;
336     var firstSource = Links.Unsync.GetTarget(upStep);
337     while (firstSource != right && firstSource != upStep)
338     {
339         upStep = firstSource;
340         firstSource = Links.Unsync.GetSource(upStep);
341     }
342     if (firstSource == right)
343     {
344         return handler(new LinkAddress<LinkIndex>(stepFrom));
345     }
346     return Constants.Continue;
347 }
348
349 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));
350
351 private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ left, LinkIndex stepFrom)
352 {
353     var upStep = stepFrom;
354     var firstTarget = Links.Unsync.GetSource(upStep);
355

```

```

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.ExtractValues();
375     var newSequence = substitution.ExtractValues();
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     }
422     return bestVariant;
423 }
424
425 private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
426 {
427     if (Options.UseGarbageCollection)
428     {
429         var sequenceElements = GetSequenceElements(sequence);
430         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
431         var sequenceLink = GetSequenceByElements(sequenceElements);
432         var newSequenceElements = GetSequenceElements(newSequence);
433         var newSequenceLink = GetSequenceByElements(newSequenceElements);
434         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)

```

```

432     {
433         if (sequenceLink != Constants.Null)
434         {
435             Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
436         }
437         Links.Unsync.MergeUsages(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.MergeUsages(sequenceLink, newSequenceLink);
455             }
456             Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
457         }
458     }
459     else
460     {
461         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
462         {
463             Links.Unsync.MergeUsages(sequence, newSequence);
464         }
465     }
466 }
467 }
468 #endregion
469 #region Delete
470
471 public void Delete(IList<LinkIndex> restrictions)
472 {
473     _sync.ExecuteWriteOperation(() =>
474     {
475         var sequence = restrictions.ExtractValues();
476         // TODO: Check all options only ones before loop execution
477         foreach (var linkToDelete in Each(sequence))
478         {
479             DeleteOneCore(linkToDelete);
480         }
481     });
482 }
483
484 private void DeleteOneCore(LinkIndex link)
485 {
486     if (Options.UseGarbageCollection)
487     {
488         var sequenceElements = GetSequenceElements(link);
489         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
490         var sequenceLink = GetSequenceByElements(sequenceElements);
491         if (Options.UseCascadeDelete || CountUsages(link) == 0)
492         {
493             if (sequenceLink != Constants.Null)
494             {
495                 Links.Unsync.Delete(sequenceLink);
496             }
497             Links.Unsync.Delete(link);
498         }
499         ClearGarbage(sequenceElementsContents.Source);
500         ClearGarbage(sequenceElementsContents.Target);
501     }
502     else
503     {
504         if (Options.UseSequenceMarker)
505         {
506             var sequenceElements = GetSequenceElements(link);
507             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 #endregion
529
530 #region Compactification
531
532 public void CompactAll()
533 {
534     _sync.ExecuteWriteOperation(() =>
535     {
536         var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
537         for (int i = 0; i < sequences.Count; i++)
538         {
539             var sequence = this.ToList(sequences[i]);
540             Compact(sequence.ConvertToRestrictionsValues());
541         }
542     });
543 }
544
545 /// <remarks>
546 /// bestVariant можно выбирать по максимальному числу использований,
547 /// но балансированный позволяет гарантировать уникальность (если есть возможность,
548 /// гарантировать его использование в других местах).
549 ///
550 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
551 /// </remarks>
552 public LinkIndex Compact(ICollection<LinkIndex> sequence)
553 {
554     return _sync.ExecuteWriteOperation(() =>
555     {
556         if (sequence.IsNullOrEmpty())
557         {
558             return Constants.Null;
559         }
560         Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
561         return CompactCore(sequence);
562     });
563 }
564
565 [MethodImpl(MethodImplOptions.AggressiveInlining)]
566 private LinkIndex CompactCore(ICollection<LinkIndex> sequence) => UpdateCore(sequence,
567     ↪ sequence);
568
569 #endregion
570
571 #region Garbage Collection
572
573 /// <remarks>
574 /// TODO0: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
575 ↪ определить извне или в унаследованном классе
576 /// </remarks>
577 [MethodImpl(MethodImplOptions.AggressiveInlining)]
578 private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
579     ↪ !Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;
580
581 private void ClearGarbage(LinkIndex link)
582 {
583     if (IsGarbage(link))
584     {
585         var contents = new Link<ulong>(Links.GetLink(link));
586         Links.Unsync.Delete(link);
587         ClearGarbage(contents.Source);
588     }
589 }

```

```

586         ClearGarbage(contents.Target);
587     }
588 }
589 #endregion
590 #region Walkers
591
592 public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
593 {
594     return _sync.ExecuteReadOperation(() =>
595     {
596         var links = Links.Unsync;
597         foreach (var part in Options.Walker.Walk(sequence))
598         {
599             if (!handler(part))
600             {
601                 return false;
602             }
603         }
604         return true;
605     });
606 }
607
608 public class Matcher : RightSequenceWalker<LinkIndex>
609 {
610     private readonly Sequences _sequences;
611     private readonly IList<LinkIndex> _patternSequence;
612     private readonly HashSet<LinkIndex> _linksInSequence;
613     private readonly HashSet<LinkIndex> _results;
614     private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
615     private readonly HashSet<LinkIndex> _readAsElements;
616     private int _filterPosition;
617
618     public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
619         ↳ HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
620         ↳ HashSet<LinkIndex> readAsElements = null)
621         : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
622     {
623         _sequences = sequences;
624         _patternSequence = patternSequence;
625         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
626             ↳ Links.Constants.Any && x != ZeroOrMany));
627         _results = results;
628         _stopableHandler = stopableHandler;
629         _readAsElements = readAsElements;
630     }
631
632     protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
633         ↳ (_readAsElements != null && _readAsElements.Contains(link)) ||
634         ↳ _linksInSequence.Contains(link);
635
636     public bool FullMatch(LinkIndex sequenceToMatch)
637     {
638         _filterPosition = 0;
639         foreach (var part in Walk(sequenceToMatch))
640         {
641             if (!FullMatchCore(part))
642             {
643                 break;
644             }
645         }
646         return _filterPosition == _patternSequence.Count;
647     }
648
649     private bool FullMatchCore(LinkIndex element)
650     {
651         if (_filterPosition == _patternSequence.Count)
652         {
653             _filterPosition = -2; // Длиннее чем нужно
654             return false;
655         }
656         if (_patternSequence[_filterPosition] != Links.Constants.Any
657             && element != _patternSequence[_filterPosition])
658         {
659             _filterPosition = -1;
660             return false; // Начинается/Продолжается иначе
661         }
662         _filterPosition++;
663         return true;
664     }

```

```

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     /// <remarks>
694     /// TODO: Add support for LinksConstants.Any
695     /// </remarks>
696     public bool PartialMatch(LinkIndex sequenceToMatch)
697     {
698         _filterPosition = -1;
699         foreach (var part in Walk(sequenceToMatch))
700         {
701             if (!PartialMatchCore(part))
702             {
703                 break;
704             }
705         }
706         return _filterPosition == _patternSequence.Count - 1;
707     }
708
709     private bool PartialMatchCore(LinkIndex element)
710     {
711         if (_filterPosition == (_patternSequence.Count - 1))
712         {
713             return false; // Нашлось
714         }
715         if (_filterPosition >= 0)
716         {
717             if (element == _patternSequence[_filterPosition + 1])
718             {
719                 _filterPosition++;
720             }
721             else
722             {
723                 _filterPosition = -1;
724             }
725         }
726         if (_filterPosition < 0)
727         {
728             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(IList<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>
766 ↵ sequencesToMatch)
767     {
768         foreach (var sequenceToMatch in sequencesToMatch)
769         {
770             if (PartialMatch(sequenceToMatch))
771             {
772                 _readAsElements.Add(sequenceToMatch);
773                 _results.Add(sequenceToMatch);
774             }
775         }
776     }
777
778     #endregion
779 }
780 }

```

./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.Data.Exceptions;
9  using Platform.Data.Sequences;
10 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
11 using Platform.Data.Doublets.Sequences.Walkers;
12 using Platform.Collections.Stacks;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     partial class Sequences
19     {
20         #region Create All Variants (Not Practical)
21
22         /// <remarks>
23         /// Number of links that is needed to generate all variants for
24         /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
25         /// </remarks>
26         public ulong[] CreateAllVariants2(ulong[] sequence)
27         {
28             return _sync.ExecuteWriteOperation(() =>
29             {
30                 if (sequence.IsNullOrEmpty())
31                 {
32                     return new ulong[0];
33                 }
34                 Links.EnsureLinkExists(sequence);
35                 if (sequence.Length == 1)

```

```

36         {
37             return sequence;
38         }
39         return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
40     });
41 }
42
43 private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
44 {
45     #if DEBUG
46         if ((stopAt - startAt) < 0)
47         {
48             throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
49                 ↳ меньше или равен stopAt");
50         }
51         #endif
52         if ((stopAt - startAt) == 0)
53         {
54             return new[] { sequence[startAt] };
55         }
56         if ((stopAt - startAt) == 1)
57         {
58             return new[] { Links.Unsync.CreateAndUpdate(sequence[startAt], sequence[stopAt])
59                 ↳ };
60         }
61         var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
62         var last = 0;
63         for (var splitter = startAt; splitter < stopAt; splitter++)
64         {
65             var left = CreateAllVariants2Core(sequence, startAt, splitter);
66             var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
67             for (var i = 0; i < left.Length; i++)
68             {
69                 for (var j = 0; j < right.Length; j++)
70                 {
71                     var variant = Links.Unsync.CreateAndUpdate(left[i], right[j]);
72                     if (variant == Constants.Null)
73                     {
74                         throw new NotImplementedException("Creation cancellation is not
75                             ↳ implemented.");
76                     }
77                     variants[last++] = variant;
78                 }
79             }
80         }
81         return variants;
82     }
83
84     public List<ulong> CreateAllVariants1(params ulong[] sequence)
85     {
86         return _sync.ExecuteWriteOperation(() =>
87         {
88             if (sequence.IsNullOrEmpty())
89             {
90                 return new List<ulong>();
91             }
92             Links.Unsync.EnsureLinkExists(sequence);
93             if (sequence.Length == 1)
94             {
95                 return new List<ulong> { sequence[0] };
96             }
97             var results = new
98                 ↳ List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
99             return CreateAllVariants1Core(sequence, results);
100         });
101     }
102
103     private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
104     {
105         if (sequence.Length == 2)
106         {
107             var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);
108             if (link == Constants.Null)
109             {
110                 throw new NotImplementedException("Creation cancellation is not
111                     ↳ implemented.");
112             }
113             results.Add(link);
114         }
115     }

```

```

109         return results;
110     }
111     var innerSequenceLength = sequence.Length - 1;
112     var innerSequence = new ulong[innerSequenceLength];
113     for (var li = 0; li < innerSequenceLength; li++)
114     {
115         var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
116         if (link == Constants.Null)
117         {
118             throw new NotImplementedException("Creation cancellation is not
119                 ↳ implemented.");
120         }
121         for (var isi = 0; isi < li; isi++)
122         {
123             innerSequence[isi] = sequence[isi];
124         }
125         innerSequence[li] = link;
126         for (var isi = li + 1; isi < innerSequenceLength; isi++)
127         {
128             innerSequence[isi] = sequence[isi + 1];
129         }
130         CreateAllVariants1Core(innerSequence, results);
131     }
132     return results;
133 }
134 #endregion
135
136 public HashSet<ulong> Each1(params ulong[] sequence)
137 {
138     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
139     Each1(link =>
140     {
141         if (!visitedLinks.Contains(link))
142         {
143             visitedLinks.Add(link); // изучить почему случаются повторы
144         }
145         return true;
146     }, sequence);
147     return visitedLinks;
148 }
149
150 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
151 {
152     if (sequence.Length == 2)
153     {
154         Links.Unsync.Each(sequence[0], sequence[1], handler);
155     }
156     else
157     {
158         var innerSequenceLength = sequence.Length - 1;
159         for (var li = 0; li < innerSequenceLength; li++)
160         {
161             var left = sequence[li];
162             var right = sequence[li + 1];
163             if (left == 0 && right == 0)
164             {
165                 continue;
166             }
167             var linkIndex = li;
168             ulong[] innerSequence = null;
169             Links.Unsync.Each(doublet =>
170             {
171                 if (innerSequence == null)
172                 {
173                     innerSequence = new ulong[innerSequenceLength];
174                     for (var isi = 0; isi < linkIndex; isi++)
175                     {
176                         innerSequence[isi] = sequence[isi];
177                     }
178                     for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
179                     {
180                         innerSequence[isi] = sequence[isi + 1];
181                     }
182                 }
183                 innerSequence[linkIndex] = doublet[Constants.IndexPart];
184                 Each1(handler, innerSequence);
185                 return Constants.Continue;
186             }, Constants.Any, left, right);

```

```

187     }
188 }
189 }
190
191 public HashSet<ulong> EachPart(params ulong[] sequence)
192 {
193     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
194     EachPartCore(link =>
195     {
196         var linkIndex = link[Constants.IndexPart];
197         if (!visitedLinks.Contains(linkIndex))
198         {
199             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
200         }
201         return Constants.Continue;
202     }, sequence);
203     return visitedLinks;
204 }
205
206 public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
207 {
208     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
209     EachPartCore(link =>
210     {
211         var linkIndex = link[Constants.IndexPart];
212         if (!visitedLinks.Contains(linkIndex))
213         {
214             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
215             return handler(new LinkAddress<LinkIndex>(linkIndex));
216         }
217         return Constants.Continue;
218     }, sequence);
219 }
220
221 private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
222     ↪ 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     }
267 }

```

```

264     });
265     //      . _x o _ .
266     //      | _ _ _ |
267     PartialStepRight(x => handler(x), sequence[0], sequence[1]);
268 }
269 else
270 {
271     throw new NotImplementedException();
272 }
273 }
274
275 private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
276 {
277     Links.Unsync.Each(Constants.Any, left, doublet =>
278     {
279         StepRight(handler, doublet, right);
280         if (left != doublet)
281         {
282             PartialStepRight(handler, doublet, right);
283         }
284         return true;
285     });
286 }
287
288 private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
289 {
290     Links.Unsync.Each(left, Constants.Any, rightStep =>
291     {
292         TryStepRightUp(handler, right, rightStep);
293         return true;
294     });
295 }
296
297 private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
298 ↪ 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;

```

```

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

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

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

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

```

```

419         if (filterPosition == sequence.Length)
420         {
421             results.Add(resultIndex);
422         }
423     }
424     if (sequence.Length >= 2)
425     {
426         StepRight(handler, sequence[0], sequence[1]);
427     }
428     var last = sequence.Length - 2;
429     for (var i = 1; i < last; i++)
430     {
431         PartialStepRight(handler, sequence[i], sequence[i + 1]);
432     }
433     if (sequence.Length >= 3)
434     {
435         StepLeft(handler, sequence[sequence.Length - 2],
436             ↪ sequence[sequence.Length - 1]);
437     }
438     return results;
439 });
440 }
441
442 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
443 {
444     return _sync.ExecuteReadOperation(() =>
445     {
446         var results = new HashSet<ulong>();
447         if (sequence.Length > 0)
448         {
449             Links.EnsureLinkExists(sequence);
450             var firstElement = sequence[0];
451             if (sequence.Length == 1)
452             {
453                 results.Add(firstElement);
454                 return results;
455             }
456             if (sequence.Length == 2)
457             {
458                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
459                 if (doublet != Constants.Null)
460                 {
461                     results.Add(doublet);
462                 }
463                 return results;
464             }
465             var matcher = new Matcher(this, sequence, results, null);
466             if (sequence.Length >= 2)
467             {
468                 StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
469             }
470             var last = sequence.Length - 2;
471             for (var i = 1; i < last; i++)
472             {
473                 PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
474                     ↪ sequence[i + 1]);
475             }
476             if (sequence.Length >= 3)
477             {
478                 StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
479                     ↪ sequence[sequence.Length - 1]);
480             }
481             return results;
482         }
483     });
484 }
485
486 public const int MaxSequenceFormatSize = 200;
487
488 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
489     ↪ => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
490
491 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
492     ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
493     ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
494     ↪ elementToString, insertComma, knownElements));
495

```

```

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

```



```

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

```

```

633         var filteredResults = new HashSet<ulong>();
634         var matcher = new Matcher(this, sequence, filteredResults, null);
635         matcher.AddAllPartialMatchedToResults(results);
636         return filteredResults;
637     }
638     return new HashSet<ulong>();
639 });
640 }
641
642 public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
643 ↪ params ulong[] sequence)
644 {
645     return _sync.ExecuteReadOperation(() =>
646     {
647         if (sequence.Length > 0)
648         {
649             Links.EnsureLinkExists(sequence);
650
651             var results = new HashSet<ulong>();
652             var filteredResults = new HashSet<ulong>();
653             var matcher = new Matcher(this, sequence, filteredResults, handler);
654             for (var i = 0; i < sequence.Length; i++)
655             {
656                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
657                 {
658                     return false;
659                 }
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);

```

```

710         AllUsagesCore(last, lastResults);
711         firstResults.IntersectWith(lastResults);
712         //for (var i = 0; i < sequence.Length; i++)
713         //    AllUsagesCore(sequence[i], results);
714         var filteredResults = new HashSet<ulong>();
715         var matcher = new Matcher(this, sequence, filteredResults, null);
716         matcher.AddAllPartialMatchedToResults(firstResults);
717         return filteredResults;
718     }
719     return new HashSet<ulong>();
720 }));
721 }
722
723 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
724     ↳ IList<ulong> sequence)
725 {
726     return _sync.ExecuteReadOperation(() =>
727     {
728         if (sequence.Count > 0)
729         {
730             Links.EnsureLinkExists(sequence);
731             var results = new HashSet<LinkIndex>();
732             //var nextResults = new HashSet<ulong>();
733             //for (var i = 0; i < sequence.Length; i++)
734             //{
735             //    AllUsagesCore(sequence[i], nextResults);
736             //    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         return new HashSet<ulong>();
766     });
767 }
768
769 // Does not work
770 //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
771 //    ↳ params ulong[] sequence)
772 //{
773 //    var visited = new HashSet<ulong>();
774 //    var results = new HashSet<ulong>();
775 //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
776 //    ↳ true; }, readAsElements);
777 //    var last = sequence.Length - 1;
778 //    for (var i = 0; i < last; i++)
779 //    {
780 //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
781 //    }
782 //    return results;
783 //}
784
785 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
786 {
787     return _sync.ExecuteReadOperation(() =>

```

```

783 {
784     if (sequence.Length > 0)
785     {
786         Links.EnsureLinkExists(sequence);
787         //var firstElement = sequence[0];
788         //if (sequence.Length == 1)
789         //{
790             //results.Add(firstElement);
791             //return results;
792         //}
793         //if (sequence.Length == 2)
794         //{
795             //var doublet = _links.SearchCore(firstElement, sequence[1]);
796             //if (doublet != Doublets.Links.Null)
797             //    results.Add(doublet);
798             //return results;
799         //}
800         //var lastElement = sequence[sequence.Length - 1];
801         //Func<ulong, bool> handler = x =>
802         //{
803             //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
804             //        results.Add(x);
805             //    return true;
806         //};
807         //if (sequence.Length >= 2)
808         //    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             //    throw new NotImplementedException(); // all sequences, containing
818             //    this element?
819         //}
820         //if (sequence.Length == 2)
821         //{
822             //    var results = new List<ulong>();
823             //    PartialStepRight(results.Add, sequence[0], sequence[1]);
824             //    return results;
825         //}
826         //var matches = new List<List<ulong>>();
827         //var last = sequence.Length - 1;
828         //for (var i = 0; i < last; i++)
829         //{
830             //    var results = new List<ulong>();
831             //    //StepRight(results.Add, sequence[i], sequence[i + 1]);
832             //    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
833             //    if (results.Count > 0)
834             //        matches.Add(results);
835             //    else
836             //        return results;
837             //    if (matches.Count == 2)
838             //    {
839                 //    var merged = new List<ulong>();
840                 //    for (var j = 0; j < matches[0].Count; j++)
841                 //        for (var k = 0; k < matches[1].Count; k++)
842                     //    CloseInnerConnections(merged.Add, matches[0][j],
843                         //        matches[1][k]);
844                 //    if (merged.Count > 0)
845                     //        matches = new List<List<ulong>> { merged };
846                 //    else
847                     //        return new List<ulong>();
848             //    }
849         //}
850         //if (matches.Count > 0)
851         //{
852             //    var usages = new HashSet<ulong>();
853             //    for (int i = 0; i < sequence.Length; i++)
854             //    {
855                 //    AllUsagesCore(sequence[i], usages);
856             //    }
857             //    //for (int i = 0; i < matches[0].Count; i++)
858             //    //    AllUsagesCore(matches[0][i], usages);
859             //    //usages.UnionWith(matches[0]);

```

```

856         return usages.ToList();
857     }
858     var firstLinkUsages = new HashSet<ulong>();
859     AllUsagesCore(sequence[0], firstLinkUsages);
860     firstLinkUsages.Add(sequence[0]);
861     //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
862     //    sequence[0] }; // or all sequences, containing this element?
863     //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
864     //    1).ToList();
865     var results = new HashSet<ulong>();
866     foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
867         firstLinkUsages, 1))
868     {
869         AllUsagesCore(match, results);
870     }
871     return results.ToList();
872 }
873
874 /// <remarks>
875 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
876 /// </remarks>
877 public HashSet<ulong> AllUsages(ulong link)
878 {
879     return _sync.ExecuteReadOperation(() =>
880     {
881         var usages = new HashSet<ulong>();
882         AllUsagesCore(link, usages);
883         return usages;
884     });
885 }
886
887 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
888 // той связи с которой начинался поиск (STTTSSSTT),
889 // причём достаточно одного бита для хранения перехода влево или вправо
890 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
891 {
892     bool handler(ulong doublet)
893     {
894         if (usages.Add(doublet))
895         {
896             AllUsagesCore(doublet, usages);
897         }
898         return true;
899     }
900     Links.Unsync.Each(link, Constants.Any, handler);
901     Links.Unsync.Each(Constants.Any, link, handler);
902 }
903
904 public HashSet<ulong> AllBottomUsages(ulong link)
905 {
906     return _sync.ExecuteReadOperation(() =>
907     {
908         var visits = new HashSet<ulong>();
909         var usages = new HashSet<ulong>();
910         AllBottomUsagesCore(link, visits, usages);
911         return usages;
912     });
913 }
914
915 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
916     usages)
917 {
918     bool handler(ulong doublet)
919     {
920         if (visits.Add(doublet))
921         {
922             AllBottomUsagesCore(doublet, visits, usages);
923         }
924         return true;
925     }
926     if (Links.Unsync.Count(Constants.Any, link) == 0)
927     {
928         usages.Add(link);
929     }
930     else

```

```

929     {
930         Links.Unsync.Each(link, Constants.Any, handler);
931         Links.Unsync.Each(Constants.Any, link, handler);
932     }
933 }
934
935 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
936 {
937     if (Options.UseSequenceMarker)
938     {
939         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
940             ↪ Options.MarkedSequenceMatcher, symbol);
941         return counter.Count();
942     }
943     else
944     {
945         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
946             ↪ symbol);
947         return counter.Count();
948     }
949 }
950
951 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
952     ↪ LinkIndex> outerHandler)
953 {
954     bool handler(ulong doublet)
955     {
956         if (usages.Add(doublet))
957         {
958             if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
959             {
960                 return false;
961             }
962             if (!AllUsagesCore1(doublet, usages, outerHandler))
963             {
964                 return false;
965             }
966         }
967         return true;
968     }
969     return Links.Unsync.Each(link, Constants.Any, handler)
970         && Links.Unsync.Each(Constants.Any, link, handler);
971 }
972
973 public void CalculateAllUsages(ulong[] totals)
974 {
975     var calculator = new AllUsagesCalculator(Links, totals);
976     calculator.Calculate();
977 }
978
979 public void CalculateAllUsages2(ulong[] totals)
980 {
981     var calculator = new AllUsagesCalculator2(Links, totals);
982     calculator.Calculate();
983 }
984
985 private class AllUsagesCalculator
986 {
987     private readonly SynchronizedLinks<ulong> _links;
988     private readonly ulong[] _totals;
989
990     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
991     {
992         _links = links;
993         _totals = totals;
994     }
995
996     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
997         ↪ CalculateCore);
998
999     private bool CalculateCore(ulong link)
1000     {
1001         if (_totals[link] == 0)
1002         {
1003             var total = 1UL;
1004             _totals[link] = total;
1005             var visitedChildren = new HashSet<ulong>();
1006             bool linkCalculator(ulong child)
1007             {

```

```

1004         if (link != child && visitedChildren.Add(child))
1005         {
1006             total += _totals[child] == 0 ? 1 : _totals[child];
1007         }
1008         return true;
1009     }
1010     _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1011     _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1012     _totals[link] = total;
1013 }
1014 return true;
1015 }
1016 }
1017
1018 private class AllUsagesCalculator2
1019 {
1020     private readonly SynchronizedLinks<ulong> _links;
1021     private readonly ulong[] _totals;
1022
1023     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1024     {
1025         _links = links;
1026         _totals = totals;
1027     }
1028
1029     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
        ↪ CalculateCore);
1030
1031     private bool IsElement(ulong link)
1032     {
1033         // _linksInSequence.Contains(link) ||
1034         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
        ↪ link;
1035     }
1036
1037     private bool CalculateCore(ulong link)
1038     {
1039         // TODO: Проработать защиту от заикливания
1040         // Основано на SequenceWalker.WalkLeft
1041         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1042         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1043         Func<ulong, bool> isElement = IsElement;
1044         void visitLeaf(ulong parent)
1045         {
1046             if (link != parent)
1047             {
1048                 _totals[parent]++;
1049             }
1050         }
1051         void visitNode(ulong parent)
1052         {
1053             if (link != parent)
1054             {
1055                 _totals[parent]++;
1056             }
1057         }
1058         var stack = new Stack();
1059         var element = link;
1060         if (isElement(element))
1061         {
1062             visitLeaf(element);
1063         }
1064         else
1065         {
1066             while (true)
1067             {
1068                 if (isElement(element))
1069                 {
1070                     if (stack.Count == 0)
1071                     {
1072                         break;
1073                     }
1074                     element = stack.Pop();
1075                     var source = getSource(element);
1076                     var target = getTarget(element);
1077                     // Обработка элемента
1078                     if (isElement(target))
1079                     {
1080                         visitLeaf(target);

```

```

1081         }
1082         if (isElement(source))
1083         {
1084             visitLeaf(source);
1085         }
1086         element = source;
1087     }
1088     else
1089     {
1090         stack.Push(element);
1091         visitNode(element);
1092         element = getTarget(element);
1093     }
1094 }
1095 }
1096 _totals[link]++;
1097 return true;
1098 }
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         _links = links;
1109         _usages = usages;
1110     }
1111
1112     public bool Collect(ulong link)
1113     {
1114         if (_usages.Add(link))
1115         {
1116             _links.Each(link, _links.Constants.Any, Collect);
1117             _links.Each(_links.Constants.Any, link, Collect);
1118         }
1119         return true;
1120     }
1121 }
1122
1123 private class AllUsagesCollector1
1124 {
1125     private readonly ILinks<ulong> _links;
1126     private readonly HashSet<ulong> _usages;
1127     private readonly ulong _continue;
1128
1129     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1130     {
1131         _links = links;
1132         _usages = usages;
1133         _continue = _links.Constants.Continue;
1134     }
1135
1136     public ulong Collect(IList<ulong> link)
1137     {
1138         var linkIndex = _links.GetIndex(link);
1139         if (_usages.Add(linkIndex))
1140         {
1141             _links.Each(Collect, _links.Constants.Any, linkIndex);
1142         }
1143         return _continue;
1144     }
1145 }
1146
1147 private class AllUsagesCollector2
1148 {
1149     private readonly ILinks<ulong> _links;
1150     private readonly BitString _usages;
1151
1152     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1153     {
1154         _links = links;
1155         _usages = usages;
1156     }
1157
1158     public bool Collect(ulong link)
1159     {
1160         if (_usages.Add((long)link))

```



```

1161         {
1162             _links.Each(link, _links.Constants.Any, Collect);
1163             _links.Each(_links.Constants.Any, link, Collect);
1164         }
1165         return true;
1166     }
1167 }
1168
1169 private class AllUsagesIntersectingCollector
1170 {
1171     private readonly SynchronizedLinks<ulong> _links;
1172     private readonly HashSet<ulong> _intersectWith;
1173     private readonly HashSet<ulong> _usages;
1174     private readonly HashSet<ulong> _enter;
1175
1176     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
↪ intersectWith, HashSet<ulong> usages)
1177     {
1178         _links = links;
1179         _intersectWith = intersectWith;
1180         _usages = usages;
1181         _enter = new HashSet<ulong>(); // защита от зацикливания
1182     }
1183
1184     public bool Collect(ulong link)
1185     {
1186         if (_enter.Add(link))
1187         {
1188             if (_intersectWith.Contains(link))
1189             {
1190                 _usages.Add(link);
1191             }
1192             _links.Unsync.Each(link, _links.Constants.Any, Collect);
1193             _links.Unsync.Each(_links.Constants.Any, link, Collect);
1194         }
1195         return true;
1196     }
1197 }
1198
1199 private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
↪ right)
1200 {
1201     TryStepLeftUp(handler, left, right);
1202     TryStepRightUp(handler, right, left);
1203 }
1204
1205 private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
↪ right)
1206 {
1207     // Direct
1208     if (left == right)
1209     {
1210         handler(new LinkAddress<LinkIndex>(left));
1211     }
1212     var doublet = Links.Unsync.SearchOrDefault(left, right);
1213     if (doublet != Constants.Null)
1214     {
1215         handler(new LinkAddress<LinkIndex>(doublet));
1216     }
1217     // Inner
1218     CloseInnerConnections(handler, left, right);
1219     // Outer
1220     StepLeft(handler, left, right);
1221     StepRight(handler, left, right);
1222     PartialStepRight(handler, left, right);
1223     PartialStepLeft(handler, left, right);
1224 }
1225
1226 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
↪ HashSet<ulong> previousMatchings, long startAt)
1227 {
1228     if (startAt >= sequence.Length) // ?
1229     {
1230         return previousMatchings;
1231     }
1232     var secondLinkUsages = new HashSet<ulong>();
1233     AllUsagesCore(sequence[startAt], secondLinkUsages);
1234     secondLinkUsages.Add(sequence[startAt]);
1235     var matchings = new HashSet<ulong>();

```

```

1236 var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
1237 //for (var i = 0; i < previousMatchings.Count; i++)
1238 foreach (var secondLinkUsage in secondLinkUsages)
1239 {
1240     foreach (var previousMatching in previousMatchings)
1241     {
1242         //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
1243         ↪ secondLinkUsage);
1244         StepRight(filler.AddFirstAndReturnConstant, previousMatching,
1245         ↪ secondLinkUsage);
1246         TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
1247         ↪ previousMatching);
1248         //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1249         ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1250         ↪ желаемым результатам.
1251         PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
1252         ↪ secondLinkUsage);
1253     }
1254 }
1255 if (matchings.Count == 0)
1256 {
1257     return matchings;
1258 }
1259 return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1260 }
1261
1262 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
1263 ↪ links, params ulong[] sequence)
1264 {
1265     if (sequence == null)
1266     {
1267         return;
1268     }
1269     for (var i = 0; i < sequence.Length; i++)
1270     {
1271         if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
1272         ↪ !links.Exists(sequence[i]))
1273         {
1274             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1275             ↪ $"patternSequence[{i}]");
1276         }
1277     }
1278 }
1279
1280 // Pattern Matching -> Key To Triggers
1281 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1282 {
1283     return _sync.ExecuteReadOperation(() =>
1284     {
1285         patternSequence = Simplify(patternSequence);
1286         if (patternSequence.Length > 0)
1287         {
1288             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1289             var uniqueSequenceElements = new HashSet<ulong>();
1290             for (var i = 0; i < patternSequence.Length; i++)
1291             {
1292                 if (patternSequence[i] != Constants.Any && patternSequence[i] !=
1293                 ↪ ZeroOrMany)
1294                 {
1295                     uniqueSequenceElements.Add(patternSequence[i]);
1296                 }
1297             }
1298             var results = new HashSet<ulong>();
1299             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1300             {
1301                 AllUsagesCore(uniqueSequenceElement, results);
1302             }
1303             var filteredResults = new HashSet<ulong>();
1304             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1305             matcher.AddAllPatternMatchedToResults(results);
1306             return filteredResults;
1307         }
1308         return new HashSet<ulong>();
1309     });
1310 }
1311
1312 // Найти все возможные связи между указанным списком связей.

```

```

1303 // Находит связи между всеми указанными связями в любом порядке.
1304 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
1305 // → несколько раз в последовательности)
1306 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1307 {
1308     return _sync.ExecuteReadOperation(() =>
1309     {
1310         var results = new HashSet<ulong>();
1311         if (linksToConnect.Length > 0)
1312         {
1313             Links.EnsureLinkExists(linksToConnect);
1314             AllUsagesCore(linksToConnect[0], results);
1315             for (var i = 1; i < linksToConnect.Length; i++)
1316             {
1317                 var next = new HashSet<ulong>();
1318                 AllUsagesCore(linksToConnect[i], next);
1319                 results.IntersectWith(next);
1320             }
1321             return results;
1322         });
1323     }
1324
1325 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1326 {
1327     return _sync.ExecuteReadOperation(() =>
1328     {
1329         var results = new HashSet<ulong>();
1330         if (linksToConnect.Length > 0)
1331         {
1332             Links.EnsureLinkExists(linksToConnect);
1333             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1334             collector1.Collect(linksToConnect[0]);
1335             var next = new HashSet<ulong>();
1336             for (var i = 1; i < linksToConnect.Length; i++)
1337             {
1338                 var collector = new AllUsagesCollector(Links.Unsync, next);
1339                 collector.Collect(linksToConnect[i]);
1340                 results.IntersectWith(next);
1341                 next.Clear();
1342             }
1343             return results;
1344         });
1345     }
1346
1347 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1348 {
1349     return _sync.ExecuteReadOperation(() =>
1350     {
1351         var results = new HashSet<ulong>();
1352         if (linksToConnect.Length > 0)
1353         {
1354             Links.EnsureLinkExists(linksToConnect);
1355             var collector1 = new AllUsagesCollector(Links, results);
1356             collector1.Collect(linksToConnect[0]);
1357             //AllUsagesCore(linksToConnect[0], results);
1358             for (var i = 1; i < linksToConnect.Length; i++)
1359             {
1360                 var next = new HashSet<ulong>();
1361                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1362                 collector.Collect(linksToConnect[i]);
1363                 //AllUsagesCore(linksToConnect[i], next);
1364                 //results.IntersectWith(next);
1365                 results = next;
1366             }
1367             return results;
1368         });
1369     }
1370
1371 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1372 {
1373     return _sync.ExecuteReadOperation(() =>
1374     {
1375         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1376         // → BitArray((int)_links.Total + 1);
1377         if (linksToConnect.Length > 0)
1378

```

```

1379     {
1380         Links.EnsureLinkExists(linksToConnect);
1381         var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1382         collector1.Collect(linksToConnect[0]);
1383         for (var i = 1; i < linksToConnect.Length; i++)
1384         {
1385             var next = new BitString((long)Links.Unsync.Count() + 1); //new
1386             ↪ BitArray((int)_links.Total + 1);
1387             var collector = new AllUsagesCollector2(Links.Unsync, next);
1388             collector.Collect(linksToConnect[i]);
1389             results = results.And(next);
1390         }
1391         return results.GetSetUInt64Indices();
1392     });
1393 }
1394
1395 private static ulong[] Simplify(ulong[] sequence)
1396 {
1397     // Считаем новый размер последовательности
1398     long newLength = 0;
1399     var zeroOrManyStepped = false;
1400     for (var i = 0; i < sequence.Length; i++)
1401     {
1402         if (sequence[i] == ZeroOrMany)
1403         {
1404             if (zeroOrManyStepped)
1405             {
1406                 continue;
1407             }
1408             zeroOrManyStepped = true;
1409         }
1410         else
1411         {
1412             //if (zeroOrManyStepped) Is it efficient?
1413             zeroOrManyStepped = false;
1414         }
1415         newLength++;
1416     }
1417     // Строим новую последовательность
1418     zeroOrManyStepped = false;
1419     var newSequence = new ulong[newLength];
1420     long j = 0;
1421     for (var i = 0; i < sequence.Length; i++)
1422     {
1423         //var current = zeroOrManyStepped;
1424         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1425         //if (current && zeroOrManyStepped)
1426         //    continue;
1427         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1428         //if (zeroOrManyStepped && newZeroOrManyStepped)
1429         //    continue;
1430         //zeroOrManyStepped = newZeroOrManyStepped;
1431         if (sequence[i] == ZeroOrMany)
1432         {
1433             if (zeroOrManyStepped)
1434             {
1435                 continue;
1436             }
1437             zeroOrManyStepped = true;
1438         }
1439         else
1440         {
1441             //if (zeroOrManyStepped) Is it efficient?
1442             zeroOrManyStepped = false;
1443         }
1444         newSequence[j++] = sequence[i];
1445     }
1446     return newSequence;
1447 }
1448
1449 public static void TestSimplify()
1450 {
1451     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
1452     ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1453     var simplifiedSequence = Simplify(sequence);
1454 }
1455
1456 public List<ulong> GetSimilarSequences() => new List<ulong>();

```

```

1456 public void Prediction()
1457 {
1458     //_links
1459     //_sequences
1460 }
1461
1462 #region From Triplets
1463
1464 //public static void DeleteSequence(Link sequence)
1465 //{
1466 //}
1467
1468 public List<ulong> CollectMatchingSequences(ulong[] links)
1469 {
1470     if (links.Length == 1)
1471     {
1472         throw new Exception("Подпоследовательности с одним элементом не
1473             ↳ поддерживаются.");
1474     }
1475     var leftBound = 0;
1476     var rightBound = links.Length - 1;
1477     var left = links[leftBound++];
1478     var right = links[rightBound--];
1479     var results = new List<ulong>();
1480     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1481     return results;
1482 }
1483
1484 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1485     ↳ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1486 {
1487     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1488     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1489     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1490     {
1491         var nextLeftLink = middleLinks[leftBound];
1492         var elements = GetRightElements(leftLink, nextLeftLink);
1493         if (leftBound <= rightBound)
1494         {
1495             for (var i = elements.Length - 1; i >= 0; i--)
1496             {
1497                 var element = elements[i];
1498                 if (element != 0)
1499                 {
1500                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
1501                         ↳ rightLink, rightBound, ref results);
1502                 }
1503             }
1504         }
1505         else
1506         {
1507             for (var i = elements.Length - 1; i >= 0; i--)
1508             {
1509                 var element = elements[i];
1510                 if (element != 0)
1511                 {
1512                     results.Add(element);
1513                 }
1514             }
1515         }
1516     }
1517     else
1518     {
1519         var nextRightLink = middleLinks[rightBound];
1520         var elements = GetLeftElements(rightLink, nextRightLink);
1521         if (leftBound <= rightBound)
1522         {
1523             for (var i = elements.Length - 1; i >= 0; i--)
1524             {
1525                 var element = elements[i];
1526                 if (element != 0)
1527                 {
1528                     CollectMatchingSequences(leftLink, leftBound, middleLinks,
1529                         ↳ elements[i], rightBound - 1, ref results);
1530                 }
1531             }
1532         }
1533     }
1534 }

```

```

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

```

```

1608     }
1609     return true;
1610 });
1611 if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1612 {
1613     result[4] = leftLink;
1614 }
1615 return result;
1616 }
1617
1618 public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1619 {
1620     var added = 0;
1621     Links.Each(Constants.Any, startLink, couple =>
1622     {
1623         if (couple != startLink)
1624         {
1625             var coupleSource = Links.GetSource(couple);
1626             if (coupleSource == leftLink)
1627             {
1628                 result[offset] = couple;
1629                 if (++added == 2)
1630                 {
1631                     return false;
1632                 }
1633             }
1634             else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1635                 ↪ == Net.And &&
1636             {
1637                 result[offset + 1] = couple;
1638                 if (++added == 2)
1639                 {
1640                     return false;
1641                 }
1642             }
1643             return true;
1644         });
1645     return added > 0;
1646 }
1647
1648 #endregion
1649
1650 #region Walkers
1651
1652 public class PatternMatcher : RightSequenceWalker<ulong>
1653 {
1654     private readonly Sequences _sequences;
1655     private readonly ulong[] _patternSequence;
1656     private readonly HashSet<LinkIndex> _linksInSequence;
1657     private readonly HashSet<LinkIndex> _results;
1658
1659     #region Pattern Match
1660
1661     enum PatternBlockType
1662     {
1663         Undefined,
1664         Gap,
1665         Elements
1666     }
1667
1668     struct PatternBlock
1669     {
1670         public PatternBlockType Type;
1671         public long Start;
1672         public long Stop;
1673     }
1674
1675     private readonly List<PatternBlock> _pattern;
1676     private int _patternPosition;
1677     private long _sequencePosition;
1678
1679     #endregion
1680
1681     public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1682         ↪ HashSet<LinkIndex> results)
1683         : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1684     {
1685         _sequences = sequences;
1686         _patternSequence = patternSequence;

```

```

1686         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1687         ↪ _sequences.Constants.Any && x != ZeroOrMany));
1688     _results = results;
1689     _pattern = CreateDetailedPattern();
1690 }
1691
1692 protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
1693 ↪ base.IsElement(link);
1694
1695 public bool PatternMatch(LinkIndex sequenceToMatch)
1696 {
1697     _patternPosition = 0;
1698     _sequencePosition = 0;
1699     foreach (var part in Walk(sequenceToMatch))
1700     {
1701         if (!PatternMatchCore(part))
1702         {
1703             break;
1704         }
1705     }
1706     return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1707     ↪ - 1 && _pattern[_patternPosition].Start == 0);
1708 }
1709
1710 private List<PatternBlock> CreateDetailedPattern()
1711 {
1712     var pattern = new List<PatternBlock>();
1713     var patternBlock = new PatternBlock();
1714     for (var i = 0; i < _patternSequence.Length; i++)
1715     {
1716         if (patternBlock.Type == PatternBlockType.Undefined)
1717         {
1718             if (_patternSequence[i] == _sequences.Constants.Any)
1719             {
1720                 patternBlock.Type = PatternBlockType.Gap;
1721                 patternBlock.Start = 1;
1722                 patternBlock.Stop = 1;
1723             }
1724             else if (_patternSequence[i] == ZeroOrMany)
1725             {
1726                 patternBlock.Type = PatternBlockType.Gap;
1727                 patternBlock.Start = 0;
1728                 patternBlock.Stop = long.MaxValue;
1729             }
1730             else
1731             {
1732                 patternBlock.Type = PatternBlockType.Elements;
1733                 patternBlock.Start = i;
1734                 patternBlock.Stop = i;
1735             }
1736         }
1737         else if (patternBlock.Type == PatternBlockType.Elements)
1738         {
1739             if (_patternSequence[i] == _sequences.Constants.Any)
1740             {
1741                 pattern.Add(patternBlock);
1742                 patternBlock = new PatternBlock
1743                 {
1744                     Type = PatternBlockType.Gap,
1745                     Start = 1,
1746                     Stop = 1
1747                 };
1748             }
1749             else if (_patternSequence[i] == ZeroOrMany)
1750             {
1751                 pattern.Add(patternBlock);
1752                 patternBlock = new PatternBlock
1753                 {
1754                     Type = PatternBlockType.Gap,
1755                     Start = 0,
1756                     Stop = long.MaxValue
1757                 };
1758             }
1759             else
1760             {
1761                 patternBlock.Stop = i;
1762             }
1763         }
1764         else // patternBlock.Type == PatternBlockType.Gap
1765         {

```



```

1763         if (_patternSequence[i] == _sequences.Constants.Any)
1764         {
1765             patternBlock.Start++;
1766             if (patternBlock.Stop < patternBlock.Start)
1767             {
1768                 patternBlock.Stop = patternBlock.Start;
1769             }
1770         }
1771         else if (_patternSequence[i] == ZeroOrMany)
1772         {
1773             patternBlock.Stop = long.MaxValue;
1774         }
1775         else
1776         {
1777             pattern.Add(patternBlock);
1778             patternBlock = new PatternBlock
1779             {
1780                 Type = PatternBlockType.Elements,
1781                 Start = i,
1782                 Stop = i
1783             };
1784         }
1785     }
1786     if (patternBlock.Type != PatternBlockType.Undefined)
1787     {
1788         pattern.Add(patternBlock);
1789     }
1790     return pattern;
1791 }
1792
1793 // match: search for regexp anywhere in text
1794 //int match(char* regexp, char* text)
1795 //{
1796 //    do
1797 //    {
1798 //    } while (*text++ != '\0');
1799 //    return 0;
1800 //}
1801
1802 // matchhere: search for regexp at beginning of text
1803 //int matchhere(char* regexp, char* text)
1804 //{
1805 //    if (regexp[0] == '\0')
1806 //        return 1;
1807 //    if (regexp[1] == '*')
1808 //        return matchstar(regexp[0], regexp + 2, text);
1809 //    if (regexp[0] == '$' && regexp[1] == '\0')
1810 //        return *text == '\0';
1811 //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1812 //        return matchhere(regexp + 1, text + 1);
1813 //    return 0;
1814 //}
1815
1816 // matchstar: search for c*regexp at beginning of text
1817 //int matchstar(int c, char* regexp, char* text)
1818 //{
1819 //    do
1820 //    {
1821 //        /* a * matches zero or more instances */
1822 //        if (matchhere(regexp, text))
1823 //            return 1;
1824 //    } while (*text != '\0' && (*text++ == c || c == '.'));
1825 //    return 0;
1826 //}
1827
1828 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
1829 //    long maximumGap)
1830 //{
1831 //    mininumGap = 0;
1832 //    maximumGap = 0;
1833 //    element = 0;
1834 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1835 //    {
1836 //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1837 //            mininumGap++;
1838 //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1839 //            maximumGap = long.MaxValue;
1840 //        else
1841 //            break;

```

```

1841 // }
1842
1843 // if (maximumGap < mininumGap)
1844 //     maximumGap = mininumGap;
1845 //}
1846
1847 private bool PatternMatchCore(LinkIndex element)
1848 {
1849     if (_patternPosition >= _pattern.Count)
1850     {
1851         _patternPosition = -2;
1852         return false;
1853     }
1854     var currentPatternBlock = _pattern[_patternPosition];
1855     if (currentPatternBlock.Type == PatternBlockType.Gap)
1856     {
1857         //var currentMatchingBlockLength = (_sequencePosition -
1858         ↪ _lastMatchedBlockPosition);
1859         if (_sequencePosition < currentPatternBlock.Start)
1860         {
1861             _sequencePosition++;
1862             return true; // Двигаемся дальше
1863         }
1864         // Это последний блок
1865         if (_pattern.Count == _patternPosition + 1)
1866         {
1867             _patternPosition++;
1868             _sequencePosition = 0;
1869             return false; // Полное соответствие
1870         }
1871         else
1872         {
1873             if (_sequencePosition > currentPatternBlock.Stop)
1874             {
1875                 return false; // Соответствие невозможно
1876             }
1877             var nextPatternBlock = _pattern[_patternPosition + 1];
1878             if (_patternSequence[nextPatternBlock.Start] == element)
1879             {
1880                 if (nextPatternBlock.Start < nextPatternBlock.Stop)
1881                 {
1882                     _patternPosition++;
1883                     _sequencePosition = 1;
1884                 }
1885                 else
1886                 {
1887                     _patternPosition += 2;
1888                     _sequencePosition = 0;
1889                 }
1890             }
1891         }
1892     }
1893     else // currentPatternBlock.Type == PatternBlockType.Elements
1894     {
1895         var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1896         if (_patternSequence[patternElementPosition] != element)
1897         {
1898             return false; // Соответствие невозможно
1899         }
1900         if (patternElementPosition == currentPatternBlock.Stop)
1901         {
1902             _patternPosition++;
1903             _sequencePosition = 0;
1904         }
1905         else
1906         {
1907             _sequencePosition++;
1908         }
1909     }
1910     return true;
1911     //if (_patternSequence[_patternPosition] != element)
1912     //    return false;
1913     //else
1914     //{
1915     //    _sequencePosition++;
1916     //    _patternPosition++;
1917     //    return true;
1918     //}
1919     //}

```

```

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

```

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

```

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

```

./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.Data.Doublets.Sequences.Frequencies.Cache;
6  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
7  using Platform.Data.Doublets.Sequences.Converters;
8  using Platform.Data.Doublets.Sequences.CriteriaMatchers;
9  using Platform.Data.Doublets.Sequences.Walkers;
10 using Platform.Data.Doublets.Sequences.Indexes;
11
12 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
        ↪ ILinks<TLink> must contain GetConstants function.
17     {
18         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
19
20         public TLink SequenceMarkerLink { get; set; }
21         public bool UseCascadeUpdate { get; set; }
22         public bool UseCascadeDelete { get; set; }
23         public bool UseIndex { get; set; } // TODO: Update Index on sequence update/delete.
24         public bool UseSequenceMarker { get; set; }
25         public bool UseCompression { get; set; }
26         public bool UseGarbageCollection { get; set; }
27         public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get; set; }
28         public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }
29
30         public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher { get; set; }
31         public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
32         public ISequenceIndex<TLink> Index { get; set; }
33         public ISequenceWalker<TLink> Walker { get; set; }
34         public bool ReadFullSequence { get; set; }
35
36         // TODO: Реализовать компактификацию при чтении
37         //public bool EnforceSingleSequenceVersionOnRead { get; set; }
38         //public bool UseRequestMarker { get; set; }
39         //public bool StoreRequestResults { get; set; }
40
41         public void InitOptions(ISynchronizedLinks<TLink> links)
42         {
43             if (UseSequenceMarker)
44             {
45                 if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
46                 {
47                     SequenceMarkerLink = links.CreatePoint();
48                 }
49                 else
50                 {
51                     if (!links.Exists(SequenceMarkerLink))
52                     {
53                         var link = links.CreatePoint();
54                         if (!_equalityComparer.Equals(link, SequenceMarkerLink))
55                         {
56                             throw new InvalidOperationException("Cannot recreate sequence marker
                                ↪ link.");
57                         }
58                     }
59                 }
60                 if (MarkedSequenceMatcher == null)
61                 {
62                     MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
                        ↪ SequenceMarkerLink);
63                 }
64             }
65             var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
66             if (UseCompression)
67             {
68                 if (LinksToSequenceConverter == null)
69                 {
70                     ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
71                     if (UseSequenceMarker)
72                     {
```

```

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

```

./Platform.Data.Doublets/Sequences/SetFiller.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences
7  {
8      public class SetFiller<TElement, TReturnConstant>
9      {
10         protected readonly ISet<TElement> _set;
11         protected readonly TReturnConstant _returnConstant;
12
13         public SetFiller(ISet<TElement> set, TReturnConstant returnConstant)
14         {
15             _set = set;
16             _returnConstant = returnConstant;
17         }
18
19         public SetFiller(ISet<TElement> set) : this(set, default) { }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public void Add(TElement element) => _set.Add(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public bool AddAndReturnTrue(TElement element)
26         {
27             _set.Add(element);
28             return true;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public bool AddFirstAndReturnTrue(ICollection<TElement> collection)
33         {
34             _set.Add(collection[0]);

```

```

35         return true;
36     }
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     public TReturnConstant AddAndReturnConstant(TElement element)
40     {
41         _set.Add(element);
42         return _returnConstant;
43     }
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> collection)
47     {
48         _set.Add(collection[0]);
49         return _returnConstant;
50     }
51 }
52 }

```

./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 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Collections.Stacks;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Sequences.Walkers
9 {
10     public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11     {
12         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
13             isElement) : base(links, stack, isElement) { }
14
15         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
16             links.IsPartialPoint) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override TLink GetNextElementAfterPop(TLink element) =>
20             Links.GetSource(element);
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetNextElementAfterPush(TLink element) =>
24             Links.GetTarget(element);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override IEnumerable<TLink> WalkContents(TLink element)
28         {
29             var parts = Links.GetLink(element);
30             var start = Links.Constants.IndexPart + 1;
31             for (var i = parts.Count - 1; i >= start; i--)
32             {
33                 var part = parts[i];
34                 if (IsElement(part))
35                 {
36                     yield return part;
37                 }
38             }
39         }
40     }
41 }

```

./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++)
49                 {
50                     var candidate = array[i];
51                     if (_equalityComparer.Equals(array[i], default))
52                     {
53                         continue;
54                     }
55                     var doubletOffset = i * 2;
56                     if (_isElement(candidate))
57                     {
58                         nextArray[doubletOffset] = candidate;
59                     }
60                     else
61                     {
62                         var link = Links.GetLink(candidate);
63                         var linkSource = Links.GetSource(link);
64                         var linkTarget = Links.GetTarget(link);
65                         nextArray[doubletOffset] = linkSource;
66                         nextArray[doubletOffset + 1] = linkTarget;
67                         if (!hasElements)
68                         {
69                             hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
70                         }
71                     }
72                 }
73             }
74             #if USEARRAYPOOL
75                 if (array.Length > 1)
76                 {
77                     ArrayPool.Free(array);
78                 }
79             #endif
80             array = nextArray;
81         }
82         while (hasElements);
83         var filledElementsCount = CountFilledElements(array);
84         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 }

```

./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 }

```



```

32     }
33 }
34 }
35 }
36 }

```

./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                 while (true)
39                 {
40                     if (IsElement(element))
41                     {
42                         if (_stack.IsEmpty)
43                         {
44                             break;
45                         }
46                         element = _stack.Pop();
47                         foreach (var output in WalkContents(element))
48                         {
49                             yield return output;
50                         }
51                         element = GetNextElementAfterPop(element);
52                     }
53                     else
54                     {
55                         _stack.Push(element);
56                         element = GetNextElementAfterPush(element);
57                     }
58                 }
59             }
60         }
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         protected abstract TLink GetNextElementAfterPop(TLink element);
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         protected abstract TLink GetNextElementAfterPush(TLink element);
70
71         [MethodImpl(MethodImplOptions.AggressiveInlining)]
72         protected abstract IEnumerable<TLink> WalkContents(TLink element);

```

```
70     }
71 }
```

./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         public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
44             ↪ _links.GetOrCreate(GetTop(), element));
45     }
46 }
```

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

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

./Platform.Data.Doublets/SynchronizedLinks.cs

```
1  using System;
2  using System.Collections.Generic;
3  using Platform.Data.Doublets;
4  using Platform.Threading.Synchronization;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets
9  {
10     /// <remarks>
11     /// TODO: Autogeneration of synchronized wrapper (decorator).
```

```

12  /// TODO: Try to unfold code of each method using IL generation for performance improvements.
13  /// TODO: Or even to unfold multiple layers of implementations.
14  /// </remarks>
15  public class SynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress>
16  {
17      public LinksConstants<TLinkAddress> Constants { get; }
18      public ISynchronization SyncRoot { get; }
19      public ILinks<TLinkAddress> Sync { get; }
20      public ILinks<TLinkAddress> Unsync { get; }
21
22      public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
        ↳ ReaderWriterLockSynchronization(), links) { }
23
24      public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
25      {
26          SyncRoot = synchronization;
27          Sync = this;
28          Unsync = links;
29          Constants = links.Constants;
30      }
31
32      public TLinkAddress Count(IList<TLinkAddress> restriction) =>
        ↳ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
33      public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
        ↳ IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
        ↳ restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
34      public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
        ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
35      public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
        ↳ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
        ↳ Unsync.Update);
36      public void Delete(IList<TLinkAddress> restrictions) =>
        ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
37
38      //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
        ↳ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
39      //{
40      //    if (restriction != null && substitution != null &&
        ↳ !substitution.EqualTo(restriction))
41      //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
        ↳ substitution, substitutedHandler, Unsync.Trigger);
42
43      //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
        ↳ substitutedHandler, Unsync.Trigger);
44      //}
45  }
46  }

```

./Platform.Data.Doublets/UInt64LinksExtensions.cs

```

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

```

```

30         {
31             return true;
32         }
33     }
34     return false;
35 }
36
37 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
    ↪ Func<Link<ulong>, bool> isElement, bool renderIndex = false, bool renderDebug =
    ↪ false)
38 {
39     var sb = new StringBuilder();
40     var visited = new HashSet<ulong>();
41     links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
    ↪ innerSb.Append(link.Index), renderIndex, renderDebug);
42     return sb.ToString();
43 }
44
45 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
    ↪ Func<Link<ulong>, bool> isElement, Action<StringBuilder, Link<ulong>> appendElement,
    ↪ bool renderIndex = false, bool renderDebug = false)
46 {
47     var sb = new StringBuilder();
48     var visited = new HashSet<ulong>();
49     links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
    ↪ renderDebug);
50     return sb.ToString();
51 }
52
53 public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
    ↪ HashSet<ulong> visited, ulong linkIndex, Func<Link<ulong>, bool> isElement,
    ↪ Action<StringBuilder, Link<ulong>> appendElement, bool renderIndex = false, bool
    ↪ renderDebug = false)
54 {
55     if (sb == null)
56     {
57         throw new ArgumentNullException(nameof(sb));
58     }
59     if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
    ↪ Constants.Itself)
60     {
61         return;
62     }
63     if (links.Exists(linkIndex))
64     {
65         if (visited.Add(linkIndex))
66         {
67             sb.Append('(');
68             var link = new Link<ulong>(links.GetLink(linkIndex));
69             if (renderIndex)
70             {
71                 sb.Append(link.Index);
72                 sb.Append(':');
73             }
74             if (link.Source == link.Index)
75             {
76                 sb.Append(link.Index);
77             }
78             else
79             {
80                 var source = new Link<ulong>(links.GetLink(link.Source));
81                 if (isElement(source))
82                 {
83                     appendElement(sb, source);
84                 }
85                 else
86                 {
87                     links.AppendStructure(sb, visited, source.Index, isElement,
    ↪ appendElement, renderIndex);
88                 }
89             }
90             sb.Append(' ');
91             if (link.Target == link.Index)
92             {
93                 sb.Append(link.Index);
94             }
95             else
96             {

```

```

97         var target = new Link<ulong>(links.GetLink(link.Target));
98         if (isElement(target))
99         {
100             appendElement(sb, target);
101         }
102         else
103         {
104             links.AppendStructure(sb, visited, target.Index, isElement,
105                 ↪ appendElement, renderIndex);
106         }
107         sb.Append(' ');
108     }
109     else
110     {
111         if (renderDebug)
112         {
113             sb.Append('*');
114         }
115         sb.Append(linkIndex);
116     }
117 }
118 else
119 {
120     if (renderDebug)
121     {
122         sb.Append('~');
123     }
124     sb.Append(linkIndex);
125 }
126 }
127 }
128 }

```

./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
86 {
87     public static readonly long Size = Structure<Transition>.Size;
88
89     public readonly ulong TransactionId;
90     public readonly Link<ulong> Before;
91     public readonly Link<ulong> After;
92     public readonly Timestamp Timestamp;
93
94     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
95     ↪ transactionId, Link<ulong> before, Link<ulong> after)
96     {
97         TransactionId = transactionId;
98         Before = before;
99         After = after;
100        Timestamp = uniqueTimestampFactory.Create();
101    }
102
103    public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
104    ↪ transactionId, Link<ulong> before)
105        : this(uniqueTimestampFactory, transactionId, before, default)
106    {
107    }
108
109    public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId)
110        : this(uniqueTimestampFactory, transactionId, default, default)
111    {
112    }
113
114    public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
115    ↪ {After}";
116 }
117
118     /// <remarks>
119     /// Другие варианты реализации транзакций (атомарности):
120     /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
121     ↪ Target)) и индексов.
122     /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
123     ↪ потребуется решить вопрос

```

```

119     со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
120     ↪ пересечениями идентификаторов.
121     ///
122     /// Где хранить промежуточный список транзакций?
123     ///
124     /// В оперативной памяти:
125     /// Минусы:
126     /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
127     /// так как нужно отдельно выделять память под список трансформаций.
128     /// 2. Выделенной оперативной памяти может не хватить, в том случае,
129     /// если транзакция использует слишком много трансформаций.
130     /// -> Можно использовать жёсткий диск для слишком длинных транзакций.
131     /// -> Максимальный размер списка трансформаций можно ограничить / задать
132     ↪ константой.
133     /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
134     ↪ создавая задержку.
135     ///
136     /// На жёстком диске:
137     /// Минусы:
138     /// 1. Длительный отклик, на запись каждой трансформации.
139     /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
140     /// -> Это может решаться упаковкой/исключением дублирующих операций.
141     /// -> Также это может решаться тем, что короткие транзакции вообще
142     /// не будут записываться в случае отката.
143     /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
144     ↪ операции (трансформации)
145     /// будут записаны в лог.
146     ///
147     /// </remarks>
148     public class Transaction : DisposableBase
149     {
150         private readonly Queue<Transition> _transitions;
151         private readonly UInt64LinksTransactionsLayer _layer;
152         public bool IsCommitted { get; private set; }
153         public bool IsReverted { get; private set; }
154
155         public Transaction(UInt64LinksTransactionsLayer layer)
156         {
157             _layer = layer;
158             if (_layer._currentTransactionId != 0)
159             {
160                 throw new NotSupportedException("Nested transactions not supported.");
161             }
162             IsCommitted = false;
163             IsReverted = false;
164             _transitions = new Queue<Transition>();
165             SetCurrentTransaction(layer, this);
166         }
167
168         public void Commit()
169         {
170             EnsureTransactionAllowsWriteOperations(this);
171             while (_transitions.Count > 0)
172             {
173                 var transition = _transitions.Dequeue();
174                 _layer._transitions.Enqueue(transition);
175             }
176             _layer._lastCommittedTransactionId = _layer._currentTransactionId;
177             IsCommitted = true;
178         }
179
180         private void Revert()
181         {
182             EnsureTransactionAllowsWriteOperations(this);
183             var transitionsToRevert = new Transition[_transitions.Count];
184             _transitions.CopyTo(transitionsToRevert, 0);
185             for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
186             {
187                 _layer.RevertTransition(transitionsToRevert[i]);
188             }
189             IsReverted = true;
190         }
191
192         public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
193             ↪ Transaction transaction)
194         {
195             layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
196             layer._currentTransactionTransitions = transaction._transitions;
197             layer._currentTransaction = transaction;
198         }
199     }

```

```

193     }
194
195     public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
196     {
197         if (transaction.IsReverted)
198         {
199             throw new InvalidOperationException("Transation is reverted.");
200         }
201         if (transaction.IsCommitted)
202         {
203             throw new InvalidOperationException("Transation is committed.");
204         }
205     }
206
207     protected override void Dispose(bool manual, bool wasDisposed)
208     {
209         if (!wasDisposed && _layer != null && !_layer.IsDisposed)
210         {
211             if (!IsCommitted && !IsReverted)
212             {
213                 Revert();
214             }
215             _layer.ResetCurrentTransation();
216         }
217     }
218 }
219
220 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
221
222 private readonly string _logAddress;
223 private readonly FileStream _log;
224 private readonly Queue<Transition> _transitions;
225 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
226 private Task _transitionsPusher;
227 private Transition _lastCommittedTransition;
228 private ulong _currentTransactionId;
229 private Queue<Transition> _currentTransactionTransitions;
230 private Transaction _currentTransaction;
231 private ulong _lastCommittedTransactionId;
232
233 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
234     : base(links)
235 {
236     if (string.IsNullOrEmpty(logAddress))
237     {
238         throw new ArgumentNullException(nameof(logAddress));
239     }
240     // В первой строке файла хранится последняя законченную транзакцию.
241     // При запуске это используется для проверки удачного закрытия файла лога.
242     // In the first line of the file the last committed transaction is stored.
243     // On startup, this is used to check that the log file is successfully closed.
244     var lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
245     var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
246     if (!lastCommittedTransition.Equals(lastWrittenTransition))
247     {
248         Dispose();
249         throw new NotSupportedException("Database is damaged, autorecovery is not
250             ↳ supported yet.");
251     }
252     if (lastCommittedTransition.Equals(default(Transition)))
253     {
254         FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
255     }
256     _lastCommittedTransition = lastCommittedTransition;
257     // TODO: Think about a better way to calculate or store this value
258     var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
259     _lastCommittedTransactionId = allTransitions.Max(x => x.TransactionId);
260     _uniqueTimestampFactory = new UniqueTimestampFactory();
261     _logAddress = logAddress;
262     _log = FileHelpers.Append(logAddress);
263     _transitions = new Queue<Transition>();
264     _transitionsPusher = new Task(TransitionsPusher);
265     _transitionsPusher.Start();
266 }
267
268 public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
269
270 public override ulong Create(IList<ulong> restrictions)
271 {

```



```

271     var createdLinkIndex = Links.Create();
272     var createdLink = new Link<ulong>(Links.GetLink(createdLinkIndex));
273     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
    ↪     default, createdLink));
274     return createdLinkIndex;
275 }
276
277 public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
278 {
279     var linkIndex = restrictions[Constants.IndexPart];
280     var beforeLink = new Link<ulong>(Links.GetLink(linkIndex));
281     linkIndex = Links.Update(restrictions, substitution);
282     var afterLink = new Link<ulong>(Links.GetLink(linkIndex));
283     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
    ↪     beforeLink, afterLink));
284     return linkIndex;
285 }
286
287 public override void Delete(IList<ulong> restrictions)
288 {
289     var link = restrictions[Constants.IndexPart];
290     var deletedLink = new Link<ulong>(Links.GetLink(link));
291     Links.Delete(link);
292     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
    ↪     deletedLink, default));
293 }
294
295 [MethodImpl(MethodImplOptions.AggressiveInlining)]
296 private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
    ↪     _transitions;
297
298 private void CommitTransition(Transition transition)
299 {
300     if (_currentTransaction != null)
301     {
302         Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
303     }
304     var transitions = GetCurrentTransitions();
305     transitions.Enqueue(transition);
306 }
307
308 private void RevertTransition(Transition transition)
309 {
310     if (transition.After.IsNull()) // Revert Deletion with Creation
311     {
312         Links.Create();
313     }
314     else if (transition.Before.IsNull()) // Revert Creation with Deletion
315     {
316         Links.Delete(transition.After.Index);
317     }
318     else // Revert Update
319     {
320         Links.Update(new[] { transition.After.Index, transition.Before.Source,
    ↪         transition.Before.Target });
321     }
322 }
323
324 private void ResetCurrentTransation()
325 {
326     _currentTransactionId = 0;
327     _currentTransactionTransitions = null;
328     _currentTransaction = null;
329 }
330
331 private void PushTransitions()
332 {
333     if (_log == null || _transitions == null)
334     {
335         return;
336     }
337     for (var i = 0; i < _transitions.Count; i++)
338     {
339         var transition = _transitions.Dequeue();
340
341         _log.Write(transition);
342         _lastCommittedTransition = transition;
343     }

```

```

344     }
345
346     private void TransitionsPusher()
347     {
348         while (!IsDisposed && _transitionsPusher != null)
349         {
350             Thread.Sleep(DefaultPushDelay);
351             PushTransitions();
352         }
353     }
354
355     public Transaction BeginTransaction() => new Transaction(this);
356
357     private void DisposeTransitions()
358     {
359         try
360         {
361             var pusher = _transitionsPusher;
362             if (pusher != null)
363             {
364                 _transitionsPusher = null;
365                 pusher.Wait();
366             }
367             if (_transitions != null)
368             {
369                 PushTransitions();
370             }
371             _log.DisposeIfPossible();
372             FileHelpers.WriteFirst(_logAddress, _lastCommittedTransition);
373         }
374         catch (Exception ex)
375         {
376             ex.Ignore();
377         }
378     }
379
380     #region DisposalBase
381
382     protected override void Dispose(bool manual, bool wasDisposed)
383     {
384         if (!wasDisposed)
385         {
386             DisposeTransitions();
387         }
388         base.Dispose(manual, wasDisposed);
389     }
390
391     #endregion
392 }
393

```

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

```

1  using Platform.Interfaces;
2  using Platform.Numbers;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Unicode
7  {
8      public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
9          ↪ IConverter<char, TLink>
10     {
11         private readonly IConverter<TLink> _addressToNumberConverter;
12         private readonly TLink _unicodeSymbolMarker;
13
14         public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
15             ↪ addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
16         {
17             _addressToNumberConverter = addressToNumberConverter;
18             _unicodeSymbolMarker = unicodeSymbolMarker;
19         }
20
21         public TLink Convert(char source)
22         {
23             var unaryNumber = _addressToNumberConverter.Convert((Integer<TLink>)source);
24             return Links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
25         }
26     }
27 }

```

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

```
1 using Platform.Data.Doublets.Sequences.Indexes;
2 using Platform.Interfaces;
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.Unicode
8 {
9     public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
10     ↪ IConverter<string, TLink>
11     {
12         private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;
13         private readonly ISequenceIndex<TLink> _index;
14         private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
15         private readonly TLink _unicodeSequenceMarker;
16
17         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
18         ↪ charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
19         ↪ TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
20         {
21             _charToUnicodeSymbolConverter = charToUnicodeSymbolConverter;
22             _index = index;
23             _listToSequenceLinkConverter = listToSequenceLinkConverter;
24             _unicodeSequenceMarker = unicodeSequenceMarker;
25         }
26
27         public TLink Convert(string source)
28         {
29             var elements = new TLink[source.Length];
30             for (int i = 0; i < source.Length; i++)
31             {
32                 elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
33             }
34             _index.Add(elements);
35             var sequence = _listToSequenceLinkConverter.Convert(elements);
36             return Links.GetOrCreate(sequence, _unicodeSequenceMarker);
37         }
38     }
39 }
```

./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 // 0 - null link
59 // 1 - nil character (0 character)
60 // ...
61 // 65536 (0(1) + 65535 = 65536 possible values)
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 public static ulong FromCharToLink(char character) => (ulong)character + 1;
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 public static char FromLinkToChar(ulong link) => (char)(link - 1);
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 public static bool IsCharLink(ulong link) => link <= MapSize;
71
72 public static string FromLinksToString(IList<ulong> linksList)
73 {
74     var sb = new StringBuilder();
75     for (int i = 0; i < linksList.Count; i++)
76     {
77         sb.Append(FromLinkToChar(linksList[i]));
78     }
79     return sb.ToString();
80 }
81
82 public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
83 {
84     var sb = new StringBuilder();
85     if (links.Exists(link))
86     {
87         StoppableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
88         ↪ x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
89         ↪ element =>
90         {
91             sb.Append(FromLinkToChar(element));
92             return true;
93         });
94     }
95     return sb.ToString();
96 }
97
98 public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
99 ↪ chars.Length);
100
101 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
102 {
103     // char array to ulong array
104     var linksSequence = new ulong[count];
105     for (var i = 0; i < count; i++)
106     {
107         linksSequence[i] = FromCharToLink(chars[i]);
108     }
109     return linksSequence;
110 }
111
112 public static ulong[] FromStringToLinkArray(string sequence)
113 {
114     // char array to ulong array
115     var linksSequence = new ulong[sequence.Length];
116     for (var i = 0; i < sequence.Length; i++)

```

```

114     {
115         linksSequence[i] = FromCharToLink(sequence[i]);
116     }
117     return linksSequence;
118 }
119
120 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
121 {
122     var result = new List<ulong[]>();
123     var offset = 0;
124     while (offset < sequence.Length)
125     {
126         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
127         var relativeLength = 1;
128         var absoluteLength = offset + relativeLength;
129         while (absoluteLength < sequence.Length &&
130             currentCategory ==
131                 CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
132         {
133             relativeLength++;
134             absoluteLength++;
135         }
136         // char array to ulong array
137         var innerSequence = new ulong[relativeLength];
138         var maxLength = offset + relativeLength;
139         for (var i = offset; i < maxLength; i++)
140         {
141             innerSequence[i - offset] = FromCharToLink(sequence[i]);
142         }
143         result.Add(innerSequence);
144         offset += relativeLength;
145     }
146     return result;
147 }
148
149 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
150 {
151     var result = new List<ulong[]>();
152     var offset = 0;
153     while (offset < array.Length)
154     {
155         var relativeLength = 1;
156         if (array[offset] <= LastCharLink)
157         {
158             var currentCategory =
159                 CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
160             var absoluteLength = offset + relativeLength;
161             while (absoluteLength < array.Length &&
162                 array[absoluteLength] <= LastCharLink &&
163                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
164                     array[absoluteLength])))
165             {
166                 relativeLength++;
167                 absoluteLength++;
168             }
169         }
170         else
171         {
172             var absoluteLength = offset + relativeLength;
173             while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
174             {
175                 relativeLength++;
176                 absoluteLength++;
177             }
178         }
179         // copy array
180         var innerSequence = new ulong[relativeLength];
181         var maxLength = offset + relativeLength;
182         for (var i = offset; i < maxLength; i++)
183         {
184             innerSequence[i - offset] = array[i];
185         }
186         result.Add(innerSequence);
187         offset += relativeLength;
188     }
189     return result;
190 }
191 }
192 }

```

./Platform.Data.Doublets/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.Doublets.Unicode
7 {
8     public class UnicodeSequenceCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
9         ↳ ICriterionMatcher<TLink>
10    {
11        private static readonly EqualityComparer<TLink> _equalityComparer =
12            ↳ EqualityComparer<TLink>.Default;
13        private readonly TLink _unicodeSequenceMarker;
14        public UnicodeSequenceCriterionMatcher(ILinks<TLink> links, TLink unicodeSequenceMarker)
15            ↳ : base(links) => _unicodeSequenceMarker = unicodeSequenceMarker;
16        public bool IsMatched(TLink link) => _equalityComparer.Equals(Links.GetTarget(link),
17            ↳ _unicodeSequenceMarker);
18    }
19 }
```

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

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

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

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

```

14     }
15 }

./Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs
1  using System;
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.Unicode
8  {
9      public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
10         ⇨ 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>
16         ⇨ numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
17         ⇨ base(links)
18         {
19             _numberToAddressConverter = numberToAddressConverter;
20             _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
21         }
22
23         public char Convert(TLink source)
24         {
25             if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
26             {
27                 throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
28                 ⇨ not a unicode symbol.");
29             }
30             return (char)(ushort)(Integer<TLink>)_numberToAddressConverter.Convert(Links.GetSource(source));
31         }
32     }
33 }

```

```

./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 }

```

./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 }

```

./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 }

```

./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     }

```

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

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using Xunit;
5  using Platform.Data.Doublets.Sequences;
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.PropertyOperators;
10 using Platform.Data.Doublets.Incrementers;
11 using Platform.Data.Doublets.Sequences.Walkers;
12 using Platform.Data.Doublets.Sequences.Indexes;
13 using Platform.Data.Doublets.Unicode;
14 using Platform.Data.Doublets.Numbers.Unary;
15 using Platform.Memory;
16 using Platform.Data.Doublets.ResizableDirectMemory;
17 using Platform.Data.Doublets.Decorators;
18 using Platform.Data.Doublets.ResizableDirectMemory.Specific;
19 using Platform.Data.Doublets.Numbers.Raw;
20 using Platform.Collections.Stacks;
21
22 namespace Platform.Data.Doublets.Tests
23 {
24     public static class OptimalVariantSequenceTests

```

```

25 {
26     private static readonly string _sequenceExample = "зеленела зелёная зелень";
27     private static readonly string _loremIpsumExample = @"Lorem ipsum dolor sit amet,
    ↳ consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore
    ↳ magna aliqua.
28 Facilisi nullam vehicula ipsum a arcu cursus vitae congue mauris.
29 Et malesuada fames ac turpis egestas sed.
30 Eget velit aliquet sagittis id consectetur purus.
31 Dignissim cras tincidunt lobortis feugiat vivamus.
32 Vitae aliquet nec ullamcorper sit.
33 Lectus quam id leo in vitae.
34 Tortor dignissim convallis aenean et tortor at risus viverra adipiscing.
35 Sed risus ultricies tristique nulla aliquet enim tortor at auctor.
36 Integer eget aliquet nibh praesent tristique.
37 Vitae congue eu consequat ac felis donec et odio.
38 Tristique et egestas quis ipsum suspendisse.
39 Suspendisse potenti nullam ac tortor vitae purus faucibus ornare.
40 Nulla facilisi etiam dignissim diam quis enim lobortis scelerisque.
41 Imperdiet proin fermentum leo vel orci.
42 In ante metus dictum at tempor commodo.
43 Nisi lacus sed viverra tellus in.
44 Quam vulputate dignissim suspendisse in.
45 Elit scelerisque mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus.
46 Gravida cum sociis natoque penatibus et magnis dis parturient.
47 Risus quis varius quam quisque id diam.
48 Congue nisi vitae suscipit tellus mauris a diam maecenas.
49 Eget nunc scelerisque viverra mauris in aliquam sem fringilla.
50 Pharetra vel turpis nunc eget lorem dolor sed viverra.
51 Mattis pellentesque id nibh tortor id aliquet.
52 Purus non enim praesent elementum facilisis leo vel.
53 Etiam sit amet nisl purus in mollis nunc sed.
54 Tortor at auctor urna nunc id cursus metus aliquam.
55 Volutpat odio facilisis mauris sit amet.
56 Turpis egestas pretium aenean pharetra magna ac placerat.
57 Fermentum dui faucibus in ornare quam viverra orci sagittis eu.
58 Porttitor leo a diam sollicitudin tempor id eu.
59 Volutpat sed cras ornare arcu dui.
60 Ut aliquam purus sit amet luctus venenatis lectus magna.
61 Aliquet risus feugiat in ante metus dictum at.
62 Mattis nunc sed blandit libero.
63 Elit pellentesque habitant morbi tristique senectus et netus.
64 Nibh sit amet commodo nulla facilisi nullam vehicula ipsum a.
65 Enim sit amet venenatis urna cursus eget nunc scelerisque viverra.
66 Amet venenatis urna cursus eget nunc scelerisque viverra mauris in.
67 Diam donec adipiscing tristique risus nec feugiat.
68 Pulvinar mattis nunc sed blandit libero volutpat.
69 Cras fermentum odio eu feugiat pretium nibh ipsum.
70 In nulla posuere sollicitudin aliquam ultrices sagittis orci a.
71 Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.
72 A iaculis at erat pellentesque.
73 Morbi blandit cursus risus at ultrices mi tempus imperdiet nulla.
74 Eget lorem dolor sed viverra ipsum nunc.
75 Leo a diam sollicitudin tempor id eu.
76 Interdum consectetur libero id faucibus nisl tincidunt eget nullam non.";
77
78 [Fact]
79 public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
80 {
81     using (var scope = new TempLinksTestScope(useSequences: false))
82     {
83         var links = scope.Links;
84         var constants = links.Constants;
85
86         links.UseUnicode();
87
88         var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
89
90         var meaningRoot = links.CreatePoint();
91         var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
92         var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
93         var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
94             ↳ constants.Itself);
95
96         var unaryNumberToAddressConverter = new
97             ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
98         var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
99         var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
100             ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
101         var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
102             ↳ frequencyPropertyMarker, frequencyMarker);
103         var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
104             ↳ frequencyPropertyOperator, frequencyIncrementer);

```

```

100     var linkToItsFrequencyNumberConverter = new
101         ↳ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
102         ↳ unaryNumberToAddressConverter);
103     var sequenceToItsLocalElementLevelsConverter = new
104         ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
105         ↳ linkToItsFrequencyNumberConverter);
106     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
107         ↳ sequenceToItsLocalElementLevelsConverter);
108
109     var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
110         ↳ Walker = new LeveledSequenceWalker<ulong>(links) });
111
112     ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
113         ↳ index, optimalVariantConverter);
114 }
115
116 [Fact]
117 public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
118 {
119     using (var scope = new TempLinksTestScope(useSequences: false))
120     {
121         var links = scope.Links;
122
123         links.UseUnicode();
124
125         var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
126
127         var totalSequenceSymbolFrequencyCounter = new
128             ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links);
129
130         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
131             ↳ totalSequenceSymbolFrequencyCounter);
132
133         var index = new
134             ↳ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
135         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
136
137         var sequenceToItsLocalElementLevelsConverter = new
138             ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
139             ↳ linkToItsFrequencyNumberConverter);
140         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
141             ↳ sequenceToItsLocalElementLevelsConverter);
142
143         var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
144             ↳ Walker = new LeveledSequenceWalker<ulong>(links) });
145
146         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
147             ↳ index, optimalVariantConverter);
148     }
149 }
150
151 private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
152     ↳ SequenceToItsLocalElementLevelsConverter<ulong>
153     ↳ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
154     ↳ OptimalVariantConverter<ulong> optimalVariantConverter)
155 {
156     index.Add(sequence);
157
158     var optimalVariant = optimalVariantConverter.Convert(sequence);
159
160     var readSequence1 = sequences.ToList(optimalVariant);
161
162     Assert.True(sequence.SequenceEqual(readSequence1));
163 }
164
165 [Fact]
166 public static void SavedSequencesOptimizationTest()
167 {
168     LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
169         ↳ (long.MaxValue + 1UL, ulong.MaxValue));
170
171     using (var memory = new HeapResizableDirectMemory())
172     using (var disposableLinks = new UInt64ResizableDirectMemoryLinks(memory,
173         ↳ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep, constants,
174         ↳ useAvlBasedIndex: false))
175     {

```

```

156     var links = new UInt64Links(disposableLinks);
157
158     var root = links.CreatePoint();
159
160     //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
161     var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
162
163     var unicodeSymbolMarker = links.GetOrCreate(root,
164         ↪ addressToNumberConverter.Convert(1));
165     var unicodeSequenceMarker = links.GetOrCreate(root,
166         ↪ addressToNumberConverter.Convert(2));
167
168     var totalSequenceSymbolFrequencyCounter = new
169         ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
170     var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
171         ↪ totalSequenceSymbolFrequencyCounter);
172     var index = new
173         ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
174     var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
175     var sequenceToItsLocalElementLevelsConverter = new
176         ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
177         ↪ linkToItsFrequencyNumberConverter);
178     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
179         ↪ sequenceToItsLocalElementLevelsConverter);
180
181     var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
182         ↪ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
183
184     var unicodeSequencesOptions = new SequencesOptions<ulong>()
185     {
186         UseSequenceMarker = true,
187         SequenceMarkerLink = unicodeSequenceMarker,
188         UseIndex = true,
189         Index = index,
190         LinksToSequenceConverter = optimalVariantConverter,
191         Walker = walker
192     };
193
194     var unicodeSequences = new Sequences.Sequences(new
195         ↪ SynchronizedLinks<ulong>(links), unicodeSequencesOptions);
196
197     // Create some sequences
198     var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
199         ↪ StringSplitOptions.RemoveEmptyEntries);
200     var arrays = strings.Select(x => x.Select(y =>
201         ↪ addressToNumberConverter.Convert(y)).ToArray()).ToArray();
202     for (int i = 0; i < arrays.Length; i++)
203     {
204         unicodeSequences.Create(arrays[i].ConvertToRestrictionsValues());
205     }
206
207     var linksCountAfterCreation = links.Count();
208
209     // get list of sequences links
210     // for each sequence link
211     //     create new sequence version
212     //     if new sequence is not the same as sequence link
213     //         delete sequence link
214     //         collect garbadge
215     //unicodeSequences.CompactAll();
216
217     //var linksCountAfterCompactification = links.Count();
218
219     //Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
220 }
221 }
222 }

```

./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

```

./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

```

```

21         File.Delete(tempFilename);
22     }
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         {
49             memoryAdapter.TestNonexistentReferences();
50         }
51
52     private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
53     {
54         var link = memoryAdapter.Create();
55         memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
56         var resultLink = _constants.Null;
57         memoryAdapter.Each(foundLink =>
58         {
59             resultLink = foundLink[_constants.IndexPart];
60             return _constants.Break;
61         }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
62         Assert.True(resultLink == link);
63         Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
64         memoryAdapter.Delete(link);
65     }
66 }

```

./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 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Linq;
5  using Xunit;
6  using Platform.Collections;
7  using Platform.Random;
8  using Platform.IO;
9  using Platform.Singletons;
10 using Platform.Data.Doublets.Sequences;
11 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
12 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
13 using Platform.Data.Doublets.Sequences.Converters;
14 using Platform.Data.Doublets.Unicode;
15
16 namespace Platform.Data.Doublets.Tests
17 {
18     public static class SequencesTests
19     {
20         private static readonly LinksConstants<ulong> _constants =
21 ↪ Default<LinksConstants<ulong>>.Instance;
22
23         static SequencesTests()
24         {
25             // Trigger static constructor to not mess with performance measurements
26             _ = BitString.GetBitMaskFromIndex(1);
27         }
28
29         [Fact]
30         public static void CreateAllVariantsTest()
31         {
32             const long sequenceLength = 8;
33
34             using (var scope = new TempLinksTestScope(useSequences: true))
35             {
36                 var links = scope.Links;
37                 var sequences = scope.Sequences;
38
39                 var sequence = new ulong[sequenceLength];
40                 for (var i = 0; i < sequenceLength; i++)
41                 {
42                     sequence[i] = links.Create();
43                 }
44
45                 var sw1 = Stopwatch.StartNew();
46                 var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
47
48                 var sw2 = Stopwatch.StartNew();
49                 var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();

```



```

49         Assert.True(results1.Count > results2.Length);
50         Assert.True(sw1.Elapsed > sw2.Elapsed);
51
52         for (var i = 0; i < sequenceLength; i++)
53         {
54             links.Delete(sequence[i]);
55         }
56
57         Assert.True(links.Count() == 0);
58     }
59 }
60
61 // [Fact]
62 // public void CUDTest()
63 // {
64 //     var tempFilename = Path.GetTempFileName();
65 //
66 //     const long sequenceLength = 8;
67 //
68 //     const ulong itself = LinksConstants.Itself;
69 //
70 //     using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
71 //         ↪ DefaultLinksSizeStep))
72 //     using (var links = new Links(memoryAdapter))
73 //     {
74 //         var sequence = new ulong[sequenceLength];
75 //         for (var i = 0; i < sequenceLength; i++)
76 //             sequence[i] = links.Create(itself, itself);
77 //
78 //         SequencesOptions o = new SequencesOptions();
79 //
80 //         TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
81 //             o.
82 //
83 //         var sequences = new Sequences(links);
84 //
85 //         var sw1 = Stopwatch.StartNew();
86 //         var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
87 //
88 //         var sw2 = Stopwatch.StartNew();
89 //         var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
90 //
91 //         Assert.True(results1.Count > results2.Length);
92 //         Assert.True(sw1.Elapsed > sw2.Elapsed);
93 //
94 //         for (var i = 0; i < sequenceLength; i++)
95 //             links.Delete(sequence[i]);
96 //     }
97 //
98 //     File.Delete(tempFilename);
99 // }
100
101 [Fact]
102 public static void AllVariantsSearchTest()
103 {
104     const long sequenceLength = 8;
105
106     using (var scope = new TempLinksTestScope(useSequences: true))
107     {
108         var links = scope.Links;
109         var sequences = scope.Sequences;
110
111         var sequence = new ulong[sequenceLength];
112         for (var i = 0; i < sequenceLength; i++)
113         {
114             sequence[i] = links.Create();
115         }
116
117         var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
118
119         // for (int i = 0; i < createResults.Length; i++)
120         //     sequences.Create(createResults[i]);
121
122         var sw0 = Stopwatch.StartNew();
123         var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
124
125         var sw1 = Stopwatch.StartNew();
126         var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
127

```

```

128     var sw2 = Stopwatch.StartNew();
129     var searchResults2 = sequences.Each1(sequence); sw2.Stop();
130
131     var sw3 = Stopwatch.StartNew();
132     var searchResults3 = sequences.Each(sequence.ConvertToRestrictionsValues());
133     ↪ sw3.Stop();
134
135     var intersection0 = createResults.Intersect(searchResults0).ToList();
136     Assert.True(intersection0.Count == searchResults0.Count);
137     Assert.True(intersection0.Count == createResults.Length);
138
139     var intersection1 = createResults.Intersect(searchResults1).ToList();
140     Assert.True(intersection1.Count == searchResults1.Count);
141     Assert.True(intersection1.Count == createResults.Length);
142
143     var intersection2 = createResults.Intersect(searchResults2).ToList();
144     Assert.True(intersection2.Count == searchResults2.Count);
145     Assert.True(intersection2.Count == createResults.Length);
146
147     var intersection3 = createResults.Intersect(searchResults3).ToList();
148     Assert.True(intersection3.Count == searchResults3.Count);
149     Assert.True(intersection3.Count == createResults.Length);
150
151     for (var i = 0; i < sequenceLength; i++)
152     {
153         links.Delete(sequence[i]);
154     }
155 }
156
157 [Fact]
158 public static void BalancedVariantSearchTest()
159 {
160     const long sequenceLength = 200;
161
162     using (var scope = new TempLinksTestScope(useSequences: true))
163     {
164         var links = scope.Links;
165         var sequences = scope.Sequences;
166
167         var sequence = new ulong[sequenceLength];
168         for (var i = 0; i < sequenceLength; i++)
169         {
170             sequence[i] = links.Create();
171         }
172
173         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
174
175         var sw1 = Stopwatch.StartNew();
176         var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
177
178         var sw2 = Stopwatch.StartNew();
179         var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
180
181         var sw3 = Stopwatch.StartNew();
182         var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
183
184         // На количестве в 200 элементов это будет занимать вечность
185         //var sw4 = Stopwatch.StartNew();
186         //var searchResults4 = sequences.Each(sequence); sw4.Stop();
187
188         Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
189
190         Assert.True(searchResults3.Count == 1 && balancedVariant ==
191             ↪ searchResults3.First());
192
193         //Assert.True(sw1.Elapsed < sw2.Elapsed);
194
195         for (var i = 0; i < sequenceLength; i++)
196         {
197             links.Delete(sequence[i]);
198         }
199     }
200
201 [Fact]
202 public static void AllPartialVariantsSearchTest()
203 {
204     const long sequenceLength = 8;

```

```

205
206 using (var scope = new TempLinksTestScope(useSequences: true))
207 {
208     var links = scope.Links;
209     var sequences = scope.Sequences;
210
211     var sequence = new ulong[sequenceLength];
212     for (var i = 0; i < sequenceLength; i++)
213     {
214         sequence[i] = links.Create();
215     }
216
217     var createResults = sequences.CreateAllVariants2(sequence);
218
219     //var createResultsStrings = createResults.Select(x => x + ": " +
220     ↪ sequences.FormatSequence(x)).ToList();
221     //Global.Trash = createResultsStrings;
222
223     var partialSequence = new ulong[sequenceLength - 2];
224
225     Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
226
227     var sw1 = Stopwatch.StartNew();
228     var searchResults1 =
229     ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
230
231     var sw2 = Stopwatch.StartNew();
232     var searchResults2 =
233     ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
234
235     //var sw3 = Stopwatch.StartNew();
236     //var searchResults3 =
237     ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
238
239     var sw4 = Stopwatch.StartNew();
240     var searchResults4 =
241     ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
242
243     //Global.Trash = searchResults3;
244
245     //var searchResults1Strings = searchResults1.Select(x => x + ": " +
246     ↪ sequences.FormatSequence(x)).ToList();
247     //Global.Trash = searchResults1Strings;
248
249     var intersection1 = createResults.Intersect(searchResults1).ToList();
250     Assert.True(intersection1.Count == createResults.Length);
251
252     var intersection2 = createResults.Intersect(searchResults2).ToList();
253     Assert.True(intersection2.Count == createResults.Length);
254
255     var intersection4 = createResults.Intersect(searchResults4).ToList();
256     Assert.True(intersection4.Count == createResults.Length);
257
258     for (var i = 0; i < sequenceLength; i++)
259     {
260         links.Delete(sequence[i]);
261     }
262 }
263
264 [Fact]
265 public static void BalancedPartialVariantsSearchTest()
266 {
267     const long sequenceLength = 200;
268
269     using (var scope = new TempLinksTestScope(useSequences: true))
270     {
271         var links = scope.Links;
272         var sequences = scope.Sequences;
273
274         var sequence = new ulong[sequenceLength];
275         for (var i = 0; i < sequenceLength; i++)
276         {
277             sequence[i] = links.Create();
278         }
279
280         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
281
282         var balancedVariant = balancedVariantConverter.Convert(sequence);

```

```

279     var partialSequence = new ulong[sequenceLength - 2];
280
281     Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
282
283     var sw1 = Stopwatch.StartNew();
284     var searchResults1 =
285         ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
286
287     var sw2 = Stopwatch.StartNew();
288     var searchResults2 =
289         ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
290
291     Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
292
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 [Fact(Skip = "Correct implementation is pending")]
303 public static void PatternMatchTest()
304 {
305     var zeroOrMany = Sequences.Sequences.ZeroOrMany;
306
307     using (var scope = new TempLinksTestScope(useSequences: true))
308     {
309         var links = scope.Links;
310         var sequences = scope.Sequences;
311
312         var e1 = links.Create();
313         var e2 = links.Create();
314
315         var sequence = new[]
316         {
317             e1, e2, e1, e2 // mama / papa
318         };
319
320         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
321
322         var balancedVariant = balancedVariantConverter.Convert(sequence);
323
324         // 1: [1]
325         // 2: [2]
326         // 3: [1,2]
327         // 4: [1,2,1,2]
328
329         var doublet = links.GetSource(balancedVariant);
330
331         var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
332
333         Assert.True(matchedSequences1.Count == 0);
334
335         var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
336
337         Assert.True(matchedSequences2.Count == 0);
338
339         var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
340
341         Assert.True(matchedSequences3.Count == 0);
342
343         var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
344
345         Assert.Contains(doublet, matchedSequences4);
346         Assert.Contains(balancedVariant, matchedSequences4);
347
348         for (var i = 0; i < sequence.Length; i++)
349         {
350             links.Delete(sequence[i]);
351         }
352     }
353 }
354
355 [Fact]
356 public static void IndexTest()
357 {

```

```

356 using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
    ↳ true }, useSequences: true))
357 {
358     var links = scope.Links;
359     var sequences = scope.Sequences;
360     var index = sequences.Options.Index;
361
362     var e1 = links.Create();
363     var e2 = links.Create();
364
365     var sequence = new[]
366     {
367         e1, e2, e1, e2 // mama / papa
368     };
369
370     Assert.False(index.MightContain(sequence));
371
372     index.Add(sequence);
373
374     Assert.True(index.MightContain(sequence));
375 }
376 }
377
378 /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%
    ↳ DO%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87
    ↳ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
379 private static readonly string _exampleText =
380     @"([english
    ↳ version] (https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))

```

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

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

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

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

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

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

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

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

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

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

```

402 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?
403
404 [![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)
405
406 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?
407
408 [![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)
409
410 Шаг назад. Возьмём опять отделённую область, которая лишь та же замкнутая линия, что ещё она
    ↳ может представлять собой? Объект? Но в чём его суть? Разве не в том, что у него есть
    ↳ граница, разделяющая внутреннее и внешнее? Допустим связь, стрелка, линия соединяет два
    ↳ объекта, как бы это выглядело?
411
412 [![белая связь, чёрная направленная
    ↳ связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png "'белая
    ↳ связь, чёрная направленная
    ↳ связь"")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png)
413
414 Допустим у нас есть смысл "'связать"' и смысл "'направления"', много ли это нам даёт? Много ли
    ↳ вариантов интерпретации? А что если уточнить, каким именно образом выполнена связь? Что если
    ↳ можно задать ей чёткий, конкретный смысл? Что это будет? Тип? Глагол? Связка? Действие?
    ↳ Трансформация? Переход из состояния в состояние? Или всё это и есть объект, суть которого в
    ↳ его конечном состоянии, если конечно конец определён направлением?
415
416 [![белая обычная и направленная связи, чёрная типизированная
    ↳ связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png "'белая
    ↳ обычная и направленная связи, чёрная типизированная
    ↳ связь"")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png)
417
418 А что если всё это время, мы смотрели на суть как бы снаружи? Можно ли взглянуть на это изнутри?
    ↳ Что будет внутри объектов? Объекты ли это? Или это связи? Может ли эта структура описать
    ↳ сама себя? Но что тогда получится, разве это не рекурсия? Может это фрактал?
419
420 [![белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная
    ↳ связь с рекурсивной внутренней
    ↳ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png
    ↳ "'белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с рекурсивной внутренней структурой"")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png)
421
422 На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
    ↳ рекурсии или фрактала?
423
424 [![белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с двойной рекурсивной внутренней
    ↳ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png
    ↳ "'белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с двойной рекурсивной внутренней структурой"")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)
425
426 Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
    ↳ Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
427
428 [![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности,
    ↳ чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://
    ↳ raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png "'белая обычная и
    ↳ направленная связи со структурой из 8 цветных элементов последовательности, чёрная
    ↳ типизированная связь со структурой из 8 цветных элементов последовательности"")] (https://raw
    ↳ .githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)
429
430 ...
431
432 [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animat
    ↳ ion-500.gif
    ↳ "'анимация"")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
    ↳ -animation-500.gif)";
433
434 private static readonly string _exampleLoremIpsumText =

```

```

435         @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
         ↳ incididunt ut labore et dolore magna aliqua.
436 Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
         ↳ consequat.";
437
438 [Fact]
439 public static void CompressionTest()
440 {
441     using (var scope = new TempLinksTestScope(useSequences: true))
442     {
443         var links = scope.Links;
444         var sequences = scope.Sequences;
445
446         var e1 = links.Create();
447         var e2 = links.Create();
448
449         var sequence = new[]
450         {
451             e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
452         };
453
454         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
455         var totalSequenceSymbolFrequencyCounter = new
456             ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
457         var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
458             ↳ totalSequenceSymbolFrequencyCounter);
459         var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
460             ↳ balancedVariantConverter, doubletFrequenciesCache);
461
462         var compressedVariant = compressingConverter.Convert(sequence);
463
464         // 1: [1]          (1->1) point
465         // 2: [2]          (2->2) point
466         // 3: [1,2]        (1->2) doublet
467         // 4: [1,2,1,2]    (3->3) doublet
468
469         Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
470         Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
471         Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
472         Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
473
474         var source = _constants.SourcePart;
475         var target = _constants.TargetPart;
476
477         Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
478         Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
479         Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
480         Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
481
482         // 4 - length of sequence
483         Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
484             ↳ == sequence[0]);
485         Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
486             ↳ == sequence[1]);
487         Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
488             ↳ == sequence[2]);
489         Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
490             ↳ == sequence[3]);
491     }
492 }
493
494 [Fact]
495 public static void CompressionEfficiencyTest()
496 {
497     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
498         ↳ StringSplitOptions.RemoveEmptyEntries);
499     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
500     var totalCharacters = arrays.Select(x => x.Length).Sum();
501
502     using (var scope1 = new TempLinksTestScope(useSequences: true))
503     using (var scope2 = new TempLinksTestScope(useSequences: true))
504     using (var scope3 = new TempLinksTestScope(useSequences: true))
505     {
506         scope1.Links.Unsync.UseUnicode();
507         scope2.Links.Unsync.UseUnicode();
508         scope3.Links.Unsync.UseUnicode();
509
510         var balancedVariantConverter1 = new
511             ↳ BalancedVariantConverter<ulong>(scope1.Links.Unsync);

```

```

503     var totalSequenceSymbolFrequencyCounter = new
504         ↳ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
505     var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
506         ↳ totalSequenceSymbolFrequencyCounter);
507     var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
508         ↳ balancedVariantConverter1, linkFrequenciesCache1,
509         ↳ doInitialFrequenciesIncrement: false);
510
511     //var compressor2 = scope2.Sequences;
512     var compressor3 = scope3.Sequences;
513
514     var constants = Default<LinksConstants<ulong>>.Instance;
515
516     var sequences = compressor3;
517     //var meaningRoot = links.CreatePoint();
518     //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
519     //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
520     //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
521         ↳ constants.Itself);
522
523     //var unaryNumberToAddressConverter = new
524         ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
525     //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
526         ↳ unaryOne);
527     //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
528         ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
529     //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
530         ↳ frequencyPropertyMarker, frequencyMarker);
531     //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
532         ↳ frequencyPropertyOperator, frequencyIncrementer);
533     //var linkToItsFrequencyNumberConverter = new
534         ↳ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
535         ↳ unaryNumberToAddressConverter);
536
537     var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
538         ↳ totalSequenceSymbolFrequencyCounter);
539
540     var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache3);
541
542     var sequenceToItsLocalElementLevelsConverter = new
543         ↳ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
544         ↳ linkToItsFrequencyNumberConverter);
545     var optimalVariantConverter = new
546         ↳ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
547         ↳ sequenceToItsLocalElementLevelsConverter);
548
549     var compressed1 = new ulong[arrays.Length];
550     var compressed2 = new ulong[arrays.Length];
551     var compressed3 = new ulong[arrays.Length];
552
553     var START = 0;
554     var END = arrays.Length;
555
556     //for (int i = START; i < END; i++)
557     //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
558
559     var initialCount1 = scope2.Links.Unsync.Count();
560
561     var sw1 = Stopwatch.StartNew();
562
563     for (int i = START; i < END; i++)
564     {
565         linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
566         compressed1[i] = compressor1.Convert(arrays[i]);
567     }
568
569     var elapsed1 = sw1.Elapsed;
570
571     var balancedVariantConverter2 = new
572         ↳ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
573
574     var initialCount2 = scope2.Links.Unsync.Count();
575
576     var sw2 = Stopwatch.StartNew();
577
578     for (int i = START; i < END; i++)
579     {

```



```

562         compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
563     }
564
565     var elapsed2 = sw2.Elapsed;
566
567     for (int i = START; i < END; i++)
568     {
569         linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
570     }
571
572     var initialCount3 = scope3.Links.Unsync.Count();
573
574     var sw3 = Stopwatch.StartNew();
575
576     for (int i = START; i < END; i++)
577     {
578         //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
579         compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
580     }
581
582     var elapsed3 = sw3.Elapsed;
583
584     Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
585         ↳ Optimal variant: {elapsed3}");
586
587     // Assert.True(elapsed1 > elapsed2);
588
589     // Checks
590     for (int i = START; i < END; i++)
591     {
592         var sequence1 = compressed1[i];
593         var sequence2 = compressed2[i];
594         var sequence3 = compressed3[i];
595
596         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
597             ↳ scope1.Links.Unsync);
598
599         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
600             ↳ scope2.Links.Unsync);
601
602         var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
603             ↳ scope3.Links.Unsync);
604
605         var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
606             ↳ link.IsPartialPoint());
607         var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
608             ↳ link.IsPartialPoint());
609         var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
610             ↳ link.IsPartialPoint());
611
612         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
613             ↳ arrays[i].Length > 3)
614         //    Assert.False(structure1 == structure2);
615         //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
616             ↳ arrays[i].Length > 3)
617         //    Assert.False(structure3 == structure2);
618
619         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
620         Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
621     }
622
623     Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
624         ↳ totalCharacters);
625     Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
626         ↳ totalCharacters);
627     Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
628         ↳ totalCharacters);
629
630     Console.WriteLine($"{{(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);

```

```

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

626
627     var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
628     var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
629     var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
630
631     var duplicates1 = duplicateCounter1.Count();
632
633     ConsoleHelpers.Debug("-----");
634
635     var duplicates2 = duplicateCounter2.Count();
636
637     ConsoleHelpers.Debug("-----");
638
639     var duplicates3 = duplicateCounter3.Count();
640
641     Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
642
643     linkFrequenciesCache1.ValidateFrequencies();
644     linkFrequenciesCache3.ValidateFrequencies();
645 }
646
647
648 [Fact]
649 public static void CompressionStabilityTest()
650 {
651     // TODO: Fix bug (do a separate test)
652     //const ulong minNumbers = 0;
653     //const ulong maxNumbers = 1000;
654
655     const ulong minNumbers = 10000;
656     const ulong maxNumbers = 12500;
657
658     var strings = new List<string>();
659
660     for (ulong i = minNumbers; i < maxNumbers; i++)
661     {
662         strings.Add(i.ToString());
663     }
664
665     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
666     var totalCharacters = arrays.Select(x => x.Length).Sum();
667
668     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
        ↳ SequencesOptions<ulong> { UseCompression = true,
        ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
669     using (var scope2 = new TempLinksTestScope(useSequences: true))
670     {
671         scope1.Links.UseUnicode();
672         scope2.Links.UseUnicode();
673
674         //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
675         var compressor1 = scope1.Sequences;
676         var compressor2 = scope2.Sequences;
677
678         var compressed1 = new ulong[arrays.Length];
679         var compressed2 = new ulong[arrays.Length];
680
681         var sw1 = Stopwatch.StartNew();
682
683         var START = 0;
684         var END = arrays.Length;
685
686         // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
687         // Stability issue starts at 10001 or 11000
688         //for (int i = START; i < END; i++)
689         //{
690             // var first = compressor1.Compress(arrays[i]);
691             // var second = compressor1.Compress(arrays[i]);
692
693             // if (first == second)
694                 // compressed1[i] = first;
695             // else
696                 // {
697                 //     // TODO: Find a solution for this case

```

```

698     //    }
699     //}
700
701     for (int i = START; i < END; i++)
702     {
703         var first = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
704         var second = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
705
706         if (first == second)
707         {
708             compressed1[i] = first;
709         }
710         else
711         {
712             // TODO: Find a solution for this case
713         }
714     }
715
716     var elapsed1 = sw1.Elapsed;
717
718     var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
719
720     var sw2 = Stopwatch.StartNew();
721
722     for (int i = START; i < END; i++)
723     {
724         var first = balancedVariantConverter.Convert(arrays[i]);
725         var second = balancedVariantConverter.Convert(arrays[i]);
726
727         if (first == second)
728         {
729             compressed2[i] = first;
730         }
731     }
732
733     var elapsed2 = sw2.Elapsed;
734
735     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
736     ↪ {elapsed2}");
737
738     Assert.True(elapsed1 > elapsed2);
739
740     // Checks
741     for (int i = START; i < END; i++)
742     {
743         var sequence1 = compressed1[i];
744         var sequence2 = compressed2[i];
745
746         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
747         {
748             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
749             ↪ scope1.Links);
750
751             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
752             ↪ scope2.Links);
753
754             //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
755             ↪ link.IsPartialPoint());
756             //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
757             ↪ link.IsPartialPoint());
758
759             //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
760             ↪ arrays[i].Length > 3)
761             //    Assert.False(structure1 == structure2);
762
763             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
764         }
765     }
766
767     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
768     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
769
770     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
771     ↪ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
772     ↪ totalCharacters}}");
773
774     Assert.True(scope1.Links.Count() <= scope2.Links.Count());
775
776     //compressor1.ValidateFrequencies();

```

```

769     }
770 }
771
772 [Fact]
773 public static void RandomNumbersCompressionQualityTest()
774 {
775     const ulong N = 500;
776
777     //const ulong minNumbers = 10000;
778     //const ulong maxNumbers = 20000;
779
780     //var strings = new List<string>();
781
782     //for (ulong i = 0; i < N; i++)
783     //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
784         ↪ maxNumbers).ToString());
785
786     var strings = new List<string>();
787     for (ulong i = 0; i < N; i++)
788     {
789         strings.Add(RandomHelpers.Default.NextUInt64().ToString());
790     }
791
792     strings = strings.Distinct().ToList();
793
794     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
795     var totalCharacters = arrays.Select(x => x.Length).Sum();
796
797     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
798         ↪ SequencesOptions<ulong> { UseCompression = true,
799         ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
800     using (var scope2 = new TempLinksTestScope(useSequences: true))
801     {
802         scope1.Links.UseUnicode();
803         scope2.Links.UseUnicode();
804
805         var compressor1 = scope1.Sequences;
806         var compressor2 = scope2.Sequences;
807
808         var compressed1 = new ulong[arrays.Length];
809         var compressed2 = new ulong[arrays.Length];
810
811         var sw1 = Stopwatch.StartNew();
812
813         var START = 0;
814         var END = arrays.Length;
815
816         for (int i = START; i < END; i++)
817         {
818             compressed1[i] = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
819         }
820
821         var elapsed1 = sw1.Elapsed;
822
823         var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
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:
834             ↪ {elapsed2}");
835
836         Assert.True(elapsed1 > elapsed2);
837
838         // Checks
839         for (int i = START; i < END; i++)
840         {
841             var sequence1 = compressed1[i];
842             var sequence2 = compressed2[i];
843
844             if (sequence1 != _constants.Null && sequence2 != _constants.Null)
845             {

```

```

844         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
845             ↪ scope1.Links);
846         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
847             ↪ scope2.Links);
848         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
849     }
850 }
851
852 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
853 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
854
855 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
856     ↪ totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
857     ↪ totalCharacters}");
858
859 // Can be worse than balanced variant
860 //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
861 //compressor1.ValidateFrequencies();
862 }
863
864 [Fact]
865 public static void AllTreeBreakDownAtSequencesCreationBugTest()
866 {
867     // Made out of AllPossibleConnectionsTest test.
868
869     //const long sequenceLength = 5; //100% bug
870     const long sequenceLength = 4; //100% bug
871     //const long sequenceLength = 3; //100% _no_bug (ok)
872
873     using (var scope = new TempLinksTestScope(useSequences: true))
874     {
875         var links = scope.Links;
876         var sequences = scope.Sequences;
877
878         var sequence = new ulong[sequenceLength];
879         for (var i = 0; i < sequenceLength; i++)
880         {
881             sequence[i] = links.Create();
882         }
883
884         var createResults = sequences.CreateAllVariants2(sequence);
885         Global.Trash = createResults;
886
887         for (var i = 0; i < sequenceLength; i++)
888         {
889             links.Delete(sequence[i]);
890         }
891     }
892 }
893
894 [Fact]
895 public static void AllPossibleConnectionsTest()
896 {
897     const long sequenceLength = 5;
898
899     using (var scope = new TempLinksTestScope(useSequences: true))
900     {
901         var links = scope.Links;
902         var sequences = scope.Sequences;
903
904         var sequence = new ulong[sequenceLength];
905         for (var i = 0; i < sequenceLength; i++)
906         {
907             sequence[i] = links.Create();
908         }
909
910         var createResults = sequences.CreateAllVariants2(sequence);
911         var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
912
913         for (var i = 0; i < 1; i++)
914         {
915             var sw1 = Stopwatch.StartNew();
916             var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
917
918

```

```

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

```

./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
            ↪ useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
            ↪ useLog) { }
19
20         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
            ↪ true, bool useSequences = false, bool useLog = false)
21         {
22             _deleteFiles = deleteFiles;
23             TempFilename = Path.GetTempFileName();
24             TempTransactionLogFilename = Path.GetTempFileName();
25             var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);
26             MemoryAdapter = useLog ? (ILinks<ulong>)new
                ↪ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
                ↪ coreMemoryAdapter;
27             Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
28             if (useSequences)
29             {
30                 Sequences = new Sequences.Sequences(Links, sequencesOptions);
31             }
32         }
33
34         protected override void Dispose(bool manual, bool wasDisposed)
35         {
36             if (!wasDisposed)
37             {
38                 Links.Unsync.DisposeIfPossible();
39                 if (_deleteFiles)
40                 {
41                     DeleteFiles();
42                 }
43             }
44         }
45
46         public void DeleteFiles()
47         {
48             File.Delete(TempFilename);
49             File.Delete(TempTransactionLogFilename);
50         }
51     }
52 }
```

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

```
1 using System.Collections.Generic;
2 using Xunit;
3 using Platform.Ranges;
4 using Platform.Numbers;
5 using Platform.Random;
6 using Platform.Setters;
7
8 namespace Platform.Data.Doublets.Tests
9 {
10     public static class TestExtensions
11     {
12         public static void TestCRUDOperations<T>(this ILinks<T> links)
13         {
14             var constants = links.Constants;
15
16             var equalityComparer = EqualityComparer<T>.Default;
17
18             // Create Link
19             Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
20 }
```

```

21     var setter = new Setter<T>(constants.Null);
22     links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
23
24     Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
25
26     var linkAddress = links.Create();
27
28     var link = new Link<T>(links.GetLink(linkAddress));
29
30     Assert.True(link.Count == 3);
31     Assert.True(equalityComparer.Equals(link.Index, linkAddress));
32     Assert.True(equalityComparer.Equals(link.Source, constants.Null));
33     Assert.True(equalityComparer.Equals(link.Target, constants.Null));
34
35     Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.One));
36
37     // Get first link
38     setter = new Setter<T>(constants.Null);
39     links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
40
41     Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
42
43     // Update link to reference itself
44     links.Update(linkAddress, linkAddress, linkAddress);
45
46     link = new Link<T>(links.GetLink(linkAddress));
47
48     Assert.True(equalityComparer.Equals(link.Source, linkAddress));
49     Assert.True(equalityComparer.Equals(link.Target, linkAddress));
50
51     // Update link to reference null (prepare for delete)
52     var updated = links.Update(linkAddress, constants.Null, constants.Null);
53
54     Assert.True(equalityComparer.Equals(updated, linkAddress));
55
56     link = new Link<T>(links.GetLink(linkAddress));
57
58     Assert.True(equalityComparer.Equals(link.Source, constants.Null));
59     Assert.True(equalityComparer.Equals(link.Target, constants.Null));
60
61     // Delete link
62     links.Delete(linkAddress);
63
64     Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
65
66     setter = new Setter<T>(constants.Null);
67     links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
68
69     Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
70 }
71
72 public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
73 {
74     // Constants
75     var constants = links.Constants;
76     var equalityComparer = EqualityComparer<T>.Default;
77
78     var h106E = new Hybrid<T>(106L, isExternal: true);
79     var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
80     var h108E = new Hybrid<T>(-108L);
81
82     Assert.Equal(106L, h106E.AbsoluteValue);
83     Assert.Equal(107L, h107E.AbsoluteValue);
84     Assert.Equal(108L, h108E.AbsoluteValue);
85
86     // Create Link (External -> External)
87     var linkAddress1 = links.Create();
88
89     links.Update(linkAddress1, h106E, h108E);
90
91     var link1 = new Link<T>(links.GetLink(linkAddress1));
92
93     Assert.True(equalityComparer.Equals(link1.Source, h106E));
94     Assert.True(equalityComparer.Equals(link1.Target, h108E));
95
96     // Create Link (Internal -> External)
97     var linkAddress2 = links.Create();
98
99     links.Update(linkAddress2, linkAddress1, h108E);
100

```



```

101     var link2 = new Link<T>(links.GetLink(linkAddress2));
102
103     Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
104     Assert.True(equalityComparer.Equals(link2.Target, h108E));
105
106     // Create Link (Internal -> Internal)
107     var linkAddress3 = links.Create();
108
109     links.Update(linkAddress3, linkAddress1, linkAddress2);
110
111     var link3 = new Link<T>(links.GetLink(linkAddress3));
112
113     Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
114     Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
115
116     // Search for created link
117     var setter1 = new Setter<T>(constants.Null);
118     links.Each(h106E, h108E, setter1.SetAndReturnFalse);
119
120     Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
121
122     // Search for nonexistent link
123     var setter2 = new Setter<T>(constants.Null);
124     links.Each(h106E, h107E, setter2.SetAndReturnFalse);
125
126     Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
127
128     // Update link to reference null (prepare for delete)
129     var updated = links.Update(linkAddress3, constants.Null, constants.Null);
130
131     Assert.True(equalityComparer.Equals(updated, linkAddress3));
132
133     link3 = new Link<T>(links.GetLink(linkAddress3));
134
135     Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
136     Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
137
138     // Delete link
139     links.Delete(linkAddress3);
140
141     Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Two));
142
143     var setter3 = new Setter<T>(constants.Null);
144     links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
145
146     Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
147 }
148
149 public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
→ links, int maximumOperationsPerCycle)
150 {
151     var comparer = Comparer<TLink>.Default;
152     for (var N = 1; N < maximumOperationsPerCycle; N++)
153     {
154         var random = new System.Random(N);
155         var created = 0;
156         var deleted = 0;
157         for (var i = 0; i < N; i++)
158         {
159             long linksCount = (Integer<TLink>)links.Count();
160             var createPoint = random.NextBoolean();
161             if (linksCount > 2 && createPoint)
162             {
163                 var linksAddressRange = new Range<ulong>(1, (ulong)linksCount);
164                 TLink source = (Integer<TLink>)random.NextUInt64(linksAddressRange);
165                 TLink target = (Integer<TLink>)random.NextUInt64(linksAddressRange);
166                 → //-V3086
167                 var resultLink = links.CreateAndUpdate(source, target);
168                 if (comparer.Compare(resultLink, (Integer<TLink>)linksCount) > 0)
169                 {
170                     created++;
171                 }
172             }
173             else
174             {
175                 links.Create();
176                 created++;
177             }
178         }
179     }

```

```

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 }

```

./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.IO;
11 using Platform.Ranges;
12 using Platform.Random;
13 using Platform.Timestamps;
14 using Platform.Reflection;
15 using Platform.Singletons;
16 using Platform.Scopes;
17 using Platform.Counters;
18 using Platform.Diagnostics;
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                 {
41                     new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeletions(100);
42                 }
43             }
44         }
45
46         [Fact]
47         public static void CascadeUpdateTest()
48         {
49             var itself = _constants.Itself;
50
51             using (var scope = new TempLinksTestScope(useLog: true))
52             {
53                 var links = scope.Links;
54
55                 var l1 = links.Create();
56                 var l2 = links.Create();
57
58                 l2 = links.Update(l2, l2, l1, l2);
59
60                 links.CreateAndUpdate(l2, itself);
61                 links.CreateAndUpdate(l2, itself);
62
63                 l2 = links.Update(l2, l1);
64
65                 links.Delete(l2);
66             }
67         }
68     }
69 }

```

```

62         Global.Trash = links.Count();
63
64         links.Unsync.DisposeIfPossible(); // Close links to access log
65
66         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
67             ↪ e.TempTransactionLogFilename);
68     }
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 [Fact]
91 public static void TransactionAutoRevertedTest()
92 {
93     // Auto Reverted (Because no commit at transaction)
94     using (var scope = new TempLinksTestScope(useLog: true))
95     {
96         var links = scope.Links;
97         var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
98         using (var transaction = transactionsLayer.BeginTransaction())
99         {
100             var l1 = links.Create();
101             var l2 = links.Create();
102
103             links.Update(l2, l2, l1, l2);
104         }
105
106         Assert.Equal(0UL, links.Count());
107
108         links.Unsync.DisposeIfPossible();
109
110         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
111             ↪ cope.TempTransactionLogFilename);
112         Assert.Single(transitions);
113     }
114
115 [Fact]
116 public static void TransactionUserCodeErrorNoDataSavedTest()
117 {
118     // User Code Error (Autoreverted), no data saved
119     var itself = _constants.Itself;
120
121     TempLinksTestScope lastScope = null;
122     try
123     {
124         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
125             ↪ useLog: true))
126         {
127             var links = scope.Links;
128             var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
129                 ↪ atorBase<ulong>)links.Unsync).Links;
130             using (var transaction = transactionsLayer.BeginTransaction())
131             {
132                 var l1 = links.CreateAndUpdate(itself, itself);
133                 var l2 = links.CreateAndUpdate(itself, itself);
134
135                 l2 = links.Update(l2, l2, l1, l2);
136
137                 links.CreateAndUpdate(l2, itself);

```

```

136         links.CreateAndUpdate(l2, itself);
137
138         //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
        ↪ tion>(scope.TempTransactionLogFilename);
139
140         l2 = links.Update(l2, l1);
141
142         links.Delete(l2);
143
144         ExceptionThrower();
145
146         transaction.Commit();
147     }
148
149     Global.Trash = links.Count();
150 }
151 }
152 catch
153 {
154     Assert.False(lastScope == null);
155
156     var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
        ↪ astScope.TempTransactionLogFilename);
157
158     Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
        ↪ transitions[0].After.IsNull());
159
160     lastScope.DeleteFiles();
161 }
162 }
163
164 [Fact]
165 public static void TransactionUserCodeErrorSomeDataSavedTest()
166 {
167     // User Code Error (Autoreverted), some data saved
168     var itself = _constants.Itself;
169
170     TempLinksTestScope lastScope = null;
171     try
172     {
173         ulong l1;
174         ulong l2;
175
176         using (var scope = new TempLinksTestScope(useLog: true))
177         {
178             var links = scope.Links;
179             l1 = links.CreateAndUpdate(itself, itself);
180             l2 = links.CreateAndUpdate(itself, itself);
181
182             l2 = links.Update(l2, l2, l1, l2);
183
184             links.CreateAndUpdate(l2, itself);
185             links.CreateAndUpdate(l2, itself);
186
187             links.Unsync.DisposeIfPossible();
188
189             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
        ↪ scope.TempTransactionLogFilename);
190         }
191
192         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
        ↪ 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
            ↪ Scope.TempTransactionLogFilename);
215
216         lastScope.DeleteFiles();
217     }
218 }
219
220 [Fact]
221 public static void TransactionCommit()
222 {
223     var itself = _constants.Itself;
224
225     var tempDatabaseFilename = Path.GetTempFileName();
226     var tempTransactionLogFilename = Path.GetTempFileName();
227
228     // Commit
229     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
            ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
            ↪ tempTransactionLogFilename))
230     using (var links = new UInt64Links(memoryAdapter))
231     {
232         using (var transaction = memoryAdapter.BeginTransaction())
233         {
234             var l1 = links.CreateAndUpdate(itself, itself);
235             var l2 = links.CreateAndUpdate(itself, itself);
236
237             Global.Trash = links.Update(l2, l2, l1, l2);
238
239             links.Delete(l1);
240
241             transaction.Commit();
242         }
243
244         Global.Trash = links.Count();
245     }
246
247     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
            ↪ sactionLogFilename);
248 }
249
250 [Fact]
251 public static void TransactionDamage()
252 {
253     var itself = _constants.Itself;
254
255     var tempDatabaseFilename = Path.GetTempFileName();
256     var tempTransactionLogFilename = Path.GetTempFileName();
257
258     // Commit
259     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
            ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
            ↪ tempTransactionLogFilename))
260     using (var links = new UInt64Links(memoryAdapter))
261     {
262         using (var transaction = memoryAdapter.BeginTransaction())
263         {
264             var l1 = links.CreateAndUpdate(itself, itself);
265             var l2 = links.CreateAndUpdate(itself, itself);
266
267             Global.Trash = links.Update(l2, l2, l1, l2);
268
269             links.Delete(l1);
270
271             transaction.Commit();
272         }
273
274         Global.Trash = links.Count();
275     }
276
277     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
            ↪ sactionLogFilename);
278
279     // Damage database
280
281     FileHelpers.WriteFirst(tempTransactionLogFilename, new
            ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));

```

```

282 // Try load damaged database
283 try
284 {
285     // TODO: Fix
286     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
287         ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
288         ↳ tempTransactionLogFilename))
289     using (var links = new UInt64Links(memoryAdapter))
290     {
291         Global.Trash = links.Count();
292     }
293 }
294 catch (NotSupportedException ex)
295 {
296     Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
297         ↳ yet.");
298 }
299
300 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
301     ↳ sactionLogFilename);
302
303 File.Delete(tempDatabaseFilename);
304 File.Delete(tempTransactionLogFilename);
305 }
306
307 [Fact]
308 public static void Bug1Test()
309 {
310     var tempDatabaseFilename = Path.GetTempFileName();
311     var tempTransactionLogFilename = Path.GetTempFileName();
312     var itself = _constants.Itself;
313
314     // User Code Error (Autoreverted), some data saved
315     try
316     {
317         ulong l1;
318         ulong l2;
319
320         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
321             ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
322             ↳ tempTransactionLogFilename))
323         using (var links = new UInt64Links(memoryAdapter))
324         {
325             l1 = links.CreateAndUpdate(itself, itself);
326             l2 = links.CreateAndUpdate(itself, itself);
327
328             l2 = links.Update(l2, l2, l1, l2);
329
330             links.CreateAndUpdate(l2, itself);
331             links.CreateAndUpdate(l2, itself);
332         }
333
334         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp_
335             ↳ TransactionLogFilename);
336
337         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
338             ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
339             ↳ tempTransactionLogFilename))
340         using (var links = new UInt64Links(memoryAdapter))
341         {
342             using (var transaction = memoryAdapter.BeginTransaction())
343             {
344                 l2 = links.Update(l2, l1);
345                 links.Delete(l2);
346                 ExceptionThrower();
347                 transaction.Commit();
348             }
349
350             Global.Trash = links.Count();
351         }
352     }
353 }
354 catch
355 {

```

```

351         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp ↵
352         ↵ TransactionLogFilename);
353     }
354     File.Delete(tempDatabaseFilename);
355     File.Delete(tempTransactionLogFilename);
356 }
357
358 private static void ExceptionThrower() => throw new InvalidOperationException();
359
360 [Fact]
361 public static void PathsTest()
362 {
363     var source = _constants.SourcePart;
364     var target = _constants.TargetPart;
365
366     using (var scope = new TempLinksTestScope())
367     {
368         var links = scope.Links;
369         var l1 = links.CreatePoint();
370         var l2 = links.CreatePoint();
371
372         var r1 = links.GetByKeys(l1, source, target, source);
373         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
374     }
375 }
376
377 [Fact]
378 public static void RecursiveStringFormattingTest()
379 {
380     using (var scope = new TempLinksTestScope(useSequences: true))
381     {
382         var links = scope.Links;
383         var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
384
385         var a = links.CreatePoint();
386         var b = links.CreatePoint();
387         var c = links.CreatePoint();
388
389         var ab = links.CreateAndUpdate(a, b);
390         var cb = links.CreateAndUpdate(c, b);
391         var ac = links.CreateAndUpdate(a, c);
392
393         a = links.Update(a, c, b);
394         b = links.Update(b, a, c);
395         c = links.Update(c, a, b);
396
397         Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
398         Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
399         Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
400
401         Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
402             ↵ "(5:(4:5 (6:5 4)) 6)");
403         Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
404             ↵ "(6:(5:(4:5 6) 6) 4)");
405         Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
406             ↵ "(4:(5:4 (6:5 4)) 6)");
407
408         // TODO: Think how to build balanced syntax tree while formatting structure (eg.
409         ↵ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)"
410
411         Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
412             ↵ "{{5}{5}{4}{6}}");
413         Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
414             ↵ "{{5}{6}{6}{4}}");
415         Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
416             ↵ "{{4}{5}{4}{6}}");
417     }
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

```

```

422     /*
423     public static void RunAllPerformanceTests()
424     {
425         try
426         {
427             links.TestLinksInSteps();
428         }
429         catch (Exception ex)
430         {
431             ex.WriteToConsole();
432         }
433
434         return;
435
436         try
437         {
438             //ThreadPool.SetMaxThreads(2, 2);
439
440             // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
↪ результат
441             // Также это дополнительно помогает в отладке
442             // Увеличивает вероятность попадания информации в кэши
443             for (var i = 0; i < 10; i++)
444             {
445                 //0 - 10 ГБ
446                 //Каждые 100 МБ срез цифр
447
448                 //links.TestGetSourceFunction();
449                 //links.TestGetSourceFunctionInParallel();
450                 //links.TestGetTargetFunction();
451                 //links.TestGetTargetFunctionInParallel();
452                 links.Create64BillionLinks();
453
454                 links.TestRandomSearchFixed();
455                 //links.Create64BillionLinksInParallel();
456                 links.TestEachFunction();
457                 //links.TestForeach();
458                 //links.TestParallelForeach();
459             }
460
461             links.TestDeletionOfAllLinks();
462
463         }
464         catch (Exception ex)
465         {
466             ex.WriteToConsole();
467         }
468     }*/
469
470     /*
471     public static void TestLinksInSteps()
472     {
473         const long gibibyte = 1024 * 1024 * 1024;
474         const long mebibyte = 1024 * 1024;
475
476         var totalLinksToCreate = gibibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
477         var linksStep = 102 * mebibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
478
479         var creationMeasurements = new List<TimeSpan>();
480         var searchMeasurements = new List<TimeSpan>();
481         var deletionMeasurements = new List<TimeSpan>();
482
483         GetBaseRandomLoopOverhead(linksStep);
484         GetBaseRandomLoopOverhead(linksStep);
485
486         var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
487
488         ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
489
490         var loops = totalLinksToCreate / linksStep;
491
492         for (int i = 0; i < loops; i++)
493         {
494             creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
495             searchMeasurements.Add(Measure(() => links.RunRandomSearches(linksStep)));
496
497             Console.WriteLine("\rC + S {0}/{1}", i + 1, loops);

```



```

498     }
499
500     ConsoleHelpers.Debug();
501
502     for (int i = 0; i < loops; i++)
503     {
504         deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
505
506         Console.WriteLine("\rD {0}/{1}", i + 1, loops);
507     }
508
509     ConsoleHelpers.Debug();
510
511     ConsoleHelpers.Debug("C S D");
512
513     for (int i = 0; i < loops; i++)
514     {
515         ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
↵ searchMeasurements[i], deletionMeasurements[i]);
516     }
517
518     ConsoleHelpers.Debug("C S D (no overhead)");
519
520     for (int i = 0; i < loops; i++)
521     {
522         ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
↵ searchMeasurements[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
523     }
524
525     ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
↵ links.Total);
526 }
527
528 private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
↵ amountToCreate)
529 {
530     for (long i = 0; i < amountToCreate; i++)
531         links.Create(0, 0);
532 }
533
534 private static TimeSpan GetBaseRandomLoopOverhead(long loops)
535 {
536     return Measure(() =>
537     {
538         ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
539         ulong result = 0;
540         for (long i = 0; i < loops; i++)
541         {
542             var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
543             var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
544
545             result += maxValue + source + target;
546         }
547         Global.Trash = result;
548     });
549 }
550 */
551
552 [Fact(Skip = "performance test")]
553 public static void GetSourceTest()
554 {
555     using (var scope = new TempLinksTestScope())
556     {
557         var links = scope.Links;
558         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
↵ Iterations);
559
560         ulong counter = 0;
561
562         //var firstLink = links.First();
563         // Создаём одну связь, из которой будет производить считывание
564         var firstLink = links.Create();
565
566         var sw = Stopwatch.StartNew();
567
568         // Тестируем саму функцию
569         for (ulong i = 0; i < Iterations; i++)
570         {
571             counter += links.GetSource(firstLink);

```

```

572     }
573
574     var elapsedTime = sw.Elapsed;
575
576     var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
577
578     // Удаляем связь, из которой производилось считывание
579     links.Delete(firstLink);
580
581     ConsoleHelpers.Debug(
582         "{0} Iterations of GetSource function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
583     }
584 }
585
586 [Fact(Skip = "performance test")]
587 public static void GetSourceInParallel()
588 {
589     using (var scope = new TempLinksTestScope())
590     {
591         var links = scope.Links;
592         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
        ↳ parallel.", Iterations);
593
594         long counter = 0;
595
596         //var firstLink = links.First();
597         var firstLink = links.Create();
598
599         var sw = Stopwatch.StartNew();
600
601         // Тестируем саму функцию
602         Parallel.For(0, Iterations, x =>
603         {
604             Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
605             //Interlocked.Increment(ref counter);
606         });
607
608         var elapsedTime = sw.Elapsed;
609
610         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
611
612         links.Delete(firstLink);
613
614         ConsoleHelpers.Debug(
615             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
616     }
617 }
618
619 [Fact(Skip = "performance test")]
620 public static void TestGetTarget()
621 {
622     using (var scope = new TempLinksTestScope())
623     {
624         var links = scope.Links;
625         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
        ↳ Iterations);
626
627         ulong counter = 0;
628
629         //var firstLink = links.First();
630         var firstLink = links.Create();
631
632         var sw = Stopwatch.StartNew();
633
634         for (ulong i = 0; i < Iterations; i++)
635         {
636             counter += links.GetTarget(firstLink);
637         }
638
639         var elapsedTime = sw.Elapsed;
640
641         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
642
643         links.Delete(firstLink);
644
645         ConsoleHelpers.Debug(
646

```

```

648         "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
649         Iterations, elapsedTime, (long)iterationsPerSecond, counter);
650     }
651 }
652
653 [Fact(Skip = "performance test")]
654 public static void TestGetTargetInParallel()
655 {
656     using (var scope = new TempLinksTestScope())
657     {
658         var links = scope.Links;
659         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
        ↳ parallel.", Iterations);
660
661         long counter = 0;
662
663         //var firstLink = links.First();
664         var firstLink = links.Create();
665
666         var sw = Stopwatch.StartNew();
667
668         Parallel.For(0, Iterations, x =>
669         {
670             Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
671             //Interlocked.Increment(ref counter);
672         });
673
674         var elapsedTime = sw.Elapsed;
675
676         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
677
678         links.Delete(firstLink);
679
680         ConsoleHelpers.Debug(
681             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
682             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
683     }
684 }
685
686 // TODO: Заполнить базу данных перед тестом
687 /*
688 [Fact]
689 public void TestRandomSearchFixed()
690 {
691     var tempFilename = Path.GetTempFileName();
692
693     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
694 ↳ DefaultLinksSizeStep))
695     {
696         long iterations = 64 * 1024 * 1024 /
697 ↳ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
698
699         ulong counter = 0;
700         var maxLink = links.Total;
701
702         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
703
704         var sw = Stopwatch.StartNew();
705
706         for (var i = iterations; i > 0; i--)
707         {
708             var source =
709 ↳ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
710             var target =
711 ↳ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
712
713             counter += links.Search(source, target);
714         }
715
716         var elapsedTime = sw.Elapsed;
717
718         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
719
720         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
721 ↳ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
722 ↳ counter);
723     }
724 }

```

```

718         File.Delete(tempFilename);
719     }*/
720
721     [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
722     public static void TestRandomSearchAll()
723     {
724         using (var scope = new TempLinksTestScope())
725         {
726             var links = scope.Links;
727             ulong counter = 0;
728
729             var maxLink = links.Count();
730
731             var iterations = links.Count();
732
733             ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
734                 ↪ links.Count());
735
736             var sw = Stopwatch.StartNew();
737
738             for (var i = iterations; i > 0; i--)
739             {
740                 var linksAddressRange = new
741                     ↪ Range<ulong>(_constants.PossibleInnerReferencesRange.Minimum, maxLink);
742
743                 var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
744                 var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
745
746                 counter += links.SearchOrDefault(source, target);
747             }
748
749             var elapsedTime = sw.Elapsed;
750
751             var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
752
753             ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
754                 ↪ Iterations per second), c: {3}",
755                 iterations, elapsedTime, (long)iterationsPerSecond, counter);
756         }
757     }
758
759     [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
760     public static void TestEach()
761     {
762         using (var scope = new TempLinksTestScope())
763         {
764             var links = scope.Links;
765
766             var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
767
768             ConsoleHelpers.Debug("Testing Each function.");
769
770             var sw = Stopwatch.StartNew();
771
772             links.Each(counter.IncrementAndReturnTrue);
773
774             var elapsedTime = sw.Elapsed;
775
776             var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
777
778             ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
779                 ↪ links per second)",
780                 counter, elapsedTime, (long)linksPerSecond);
781         }
782     }
783
784     /*
785     [Fact]
786     public static void TestForeach()
787     {
788         var tempFilename = Path.GetTempFileName();
789
790         using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
791             ↪ DefaultLinksSizeStep))
792         {
793             ulong counter = 0;
794
795             ConsoleHelpers.Debug("Testing foreach through links.");
796         }
797     }
798     */

```

```

793         var sw = Stopwatch.StartNew();
794
795         //foreach (var link in links)
796         //{
797             counter++;
798         //}
799
800         var elapsedTime = sw.Elapsed;
801
802         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
803
804         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
↪ links per second)", counter, elapsedTime, (long)linksPerSecond);
805     }
806
807     File.Delete(tempFilename);
808 }
809 */
810
811 /*
812 [Fact]
813 public static void TestParallelForeach()
814 {
815     var tempFilename = Path.GetTempFileName();
816
817     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪ DefaultLinksSizeStep))
818     {
819         long counter = 0;
820
821         ConsoleHelpers.Debug("Testing parallel foreach through links.");
822
823         var sw = Stopwatch.StartNew();
824
825         //Parallel.ForEach((IEnumerable<ulong>)links, x =>
826         //{
827             // Interlocked.Increment(ref counter);
828         //});
829
830         var elapsedTime = sw.Elapsed;
831
832         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
833
834         ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↪ {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
835     }
836
837     File.Delete(tempFilename);
838 }
839 */
840
841 [Fact(Skip = "performance test")]
842 public static void Create64BillionLinks()
843 {
844     using (var scope = new TempLinksTestScope())
845     {
846         var links = scope.Links;
847         var linksBeforeTest = links.Count();
848
849         long linksToCreate = 64 * 1024 * 1024 /
↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
850
851         ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
852
853         var elapsedTime = Performance.Measure(() =>
854         {
855             for (long i = 0; i < linksToCreate; i++)
856             {
857                 links.Create();
858             }
859         });
860
861         var linksCreated = links.Count() - linksBeforeTest;
862         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
863
864         ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
865
866         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
↪ linksCreated, elapsedTime,

```

```

868         (long)linksPerSecond);
869     }
870 }
871
872 [Fact(Skip = "performance test")]
873 public static void Create64BillionLinksInParallel()
874 {
875     using (var scope = new TempLinksTestScope())
876     {
877         var links = scope.Links;
878         var linksBeforeTest = links.Count();
879
880         var sw = Stopwatch.StartNew();
881
882         long linksToCreate = 64 * 1024 * 1024 /
883             ↪ 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)",
895             ↪ linksCreated, elapsedTime,
896             (long)linksPerSecond);
897     }
898 }
899
900 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
901 public static void TestDeletionOfAllLinks()
902 {
903     using (var scope = new TempLinksTestScope())
904     {
905         var links = scope.Links;
906         var linksBeforeTest = links.Count();
907
908         ConsoleHelpers.Debug("Deleting all links");
909
910         var elapsedTime = Performance.Measure(links.DeleteAll);
911
912         var linksDeleted = linksBeforeTest - links.Count();
913         var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
914
915         ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
916             ↪ linksDeleted, elapsedTime,
917             (long)linksPerSecond);
918     }
919 }
920
921 #endregion
922 }

```

./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);

```

```

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

```

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

```

1  using Xunit;
2  using Platform.Interfaces;
3  using Platform.Memory;
4  using Platform.Reflection;
5  using Platform.Scopes;
6  using Platform.Data.Doublets.Incrementers;
7  using Platform.Data.Doublets.Numbers.Raw;
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 }

```

```

48 private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
    ↳ meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
    ↳ numberToAddressConverter)
49 {
50     var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
51     var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
    ↳ addressToNumberConverter, unicodeSymbolMarker);
52     var originalCharacter = 'H';
53     var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
54     var unicodeSymbolCriterionMatcher = new UnicodeSymbolCriterionMatcher<ulong>(links,
    ↳ unicodeSymbolMarker);
55     var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
    ↳ numberToAddressConverter, unicodeSymbolCriterionMatcher);
56     var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
57     Assert.Equal(originalCharacter, resultingCharacter);
58 }
59
60 [Fact]
61 public static void StringAndUnicodeSequenceConvertersTest()
62 {
63     using (var scope = new TempLinksTestScope())
64     {
65         var links = scope.Links;
66
67         var itself = links.Constants.Itself;
68
69         var meaningRoot = links.CreatePoint();
70         var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
71         var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
72         var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
73         var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
74         var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
75
76         var powerOf2ToUnaryNumberConverter = new
    ↳ PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
77         var addressToUnaryNumberConverter = new
    ↳ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
78         var charToUnicodeSymbolConverter = new
    ↳ CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
    ↳ unicodeSymbolMarker);
79
80         var unaryNumberToAddressConverter = new
    ↳ UnaryNumberToAddressOrOperationConverter<ulong>(links,
    ↳ powerOf2ToUnaryNumberConverter);
81         var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
82         var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
    ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
83         var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
    ↳ frequencyPropertyMarker, frequencyMarker);
84         var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
    ↳ frequencyPropertyOperator, frequencyIncrementer);
85         var linkToItsFrequencyNumberConverter = new
    ↳ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
    ↳ unaryNumberToAddressConverter);
86         var sequenceToItsLocalElementLevelsConverter = new
    ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
    ↳ linkToItsFrequencyNumberConverter);
87         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
    ↳ sequenceToItsLocalElementLevelsConverter);
88
89         var stringToUnicodeSequenceConverter = new
    ↳ StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
    ↳ index, optimalVariantConverter, unicodeSequenceMarker);
90
91         var originalString = "Hello";
92
93         var unicodeSequenceLink =
    ↳ stringToUnicodeSequenceConverter.Convert(originalString);
94
95         var unicodeSymbolCriterionMatcher = new
    ↳ UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
96         var unicodeSymbolToCharConverter = new
    ↳ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
    ↳ unicodeSymbolCriterionMatcher);
97
98         var unicodeSequenceCriterionMatcher = new
    ↳ UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
99

```



```

100     var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
    ↪     unicodeSymbolCriterionMatcher.IsMatched);
101
102     var unicodeSequenceToStringConverter = new
    ↪     UnicodeSequenceToStringConverter<ulong>(links,
    ↪     unicodeSequenceCriterionMatcher, sequenceWalker,
    ↪     unicodeSymbolToCharConverter);
103
104     var resultingString =
    ↪     unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
105
106     Assert.Equal(originalString, resultingString);
107 }
108 }
109 }
110 }

```

Index

- ./Platform.Data.Doublets.Tests/ComparisonTests.cs, 143
- ./Platform.Data.Doublets.Tests/EqualityTests.cs, 144
- ./Platform.Data.Doublets.Tests/GenericLinksTests.cs, 145
- ./Platform.Data.Doublets.Tests/LinksConstantsTests.cs, 146
- ./Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs, 146
- ./Platform.Data.Doublets.Tests/ReadSequenceTests.cs, 149
- ./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs, 150
- ./Platform.Data.Doublets.Tests/ScopeTests.cs, 151
- ./Platform.Data.Doublets.Tests/SequencesTests.cs, 152
- ./Platform.Data.Doublets.Tests/TempLinksTestScope.cs, 166
- ./Platform.Data.Doublets.Tests/TestExtensions.cs, 167
- ./Platform.Data.Doublets.Tests/UInt64LinksTests.cs, 170
- ./Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs, 182
- ./Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs, 183
- ./Platform.Data.Doublets/Decorators/LinksCascadeUniquenessAndUsagesResolver.cs, 1
- ./Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.cs, 1
- ./Platform.Data.Doublets/Decorators/LinksDecoratorBase.cs, 1
- ./Platform.Data.Doublets/Decorators/LinksDisposableDecoratorBase.cs, 2
- ./Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.cs, 3
- ./Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.cs, 3
- ./Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.cs, 4
- ./Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.cs, 4
- ./Platform.Data.Doublets/Decorators/LinksUniquenessResolver.cs, 5
- ./Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs, 5
- ./Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs, 5
- ./Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs, 6
- ./Platform.Data.Doublets/Decorators/UInt64Links.cs, 6
- ./Platform.Data.Doublets/Decorators/UniLinks.cs, 7
- ./Platform.Data.Doublets/Doublet.cs, 12
- ./Platform.Data.Doublets/DoubletComparer.cs, 12
- ./Platform.Data.Doublets/Hybrid.cs, 13
- ./Platform.Data.Doublets/ILinks.cs, 14
- ./Platform.Data.Doublets/ILinksExtensions.cs, 15
- ./Platform.Data.Doublets/ISynchronizedLinks.cs, 27
- ./Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs, 26
- ./Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs, 26
- ./Platform.Data.Doublets/Link.cs, 27
- ./Platform.Data.Doublets/LinkExtensions.cs, 30
- ./Platform.Data.Doublets/LinksOperatorBase.cs, 30
- ./Platform.Data.Doublets/Numbers/Raw/AddressToRawNumberConverter.cs, 30
- ./Platform.Data.Doublets/Numbers/Raw/RawNumberToAddressConverter.cs, 30
- ./Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs, 30
- ./Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConverter.cs, 31
- ./Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs, 32
- ./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs, 32
- ./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs, 33
- ./Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 34
- ./Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 34
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksAvlBalancedTreeMethodsBase.cs, 35
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSizeBalancedTreeMethodsBase.cs, 39
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesAvlBalancedTreeMethods.cs, 42
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesSizeBalancedTreeMethods.cs, 43
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsAvlBalancedTreeMethods.cs, 44
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsSizeBalancedTreeMethods.cs, 46
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinks.cs, 53
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinksBase.cs, 46
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/UnusedLinksListMethods.cs, 55
- ./Platform.Data.Doublets/ResizableDirectMemory/ILinksListMethods.cs, 55
- ./Platform.Data.Doublets/ResizableDirectMemory/ILinksTreeMethods.cs, 56
- ./Platform.Data.Doublets/ResizableDirectMemory/LinksHeader.cs, 56
- ./Platform.Data.Doublets/ResizableDirectMemory/RawLink.cs, 56
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs, 56
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs, 58
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs, 59
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.cs, 61
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs, 61

./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs, 63
./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64ResizableDirectMemoryLinks.cs, 64
./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64UnusedLinksListMethods.cs, 65
./Platform.Data.Doublets/Sequences/ArrayExtensions.cs, 66
./Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 66
./Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 67
./Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 70
./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 70
./Platform.Data.Doublets/Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 71
./Platform.Data.Doublets/Sequences/CreteriaMatchers/DefaultSequenceElementCriterionMatcher.cs, 72
./Platform.Data.Doublets/Sequences/CreteriaMatchers/MarkedSequenceCriterionMatcher.cs, 72
./Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 72
./Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 73
./Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 73
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 76
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 77
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToltsFrequencyValueConverter.cs, 78
./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 78
./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 78
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 79
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 79
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 80
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 80
./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 81
./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 82
./Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 82
./Platform.Data.Doublets/Sequences/IListExtensions.cs, 82
./Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 83
./Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 83
./Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 84
./Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 84
./Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 85
./Platform.Data.Doublets/Sequences/Indexes/Unindex.cs, 86
./Platform.Data.Doublets/Sequences/ListFiller.cs, 86
./Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 97
./Platform.Data.Doublets/Sequences/Sequences.cs, 87
./Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 123
./Platform.Data.Doublets/Sequences/SequencesOptions.cs, 123
./Platform.Data.Doublets/Sequences/SetFiller.cs, 125
./Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 126
./Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 126
./Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 126
./Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 128
./Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 129
./Platform.Data.Doublets/Stacks/Stack.cs, 130
./Platform.Data.Doublets/Stacks/StackExtensions.cs, 130
./Platform.Data.Doublets/SynchronizedLinks.cs, 130
./Platform.Data.Doublets/UInt64LinksExtensions.cs, 131
./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 133
./Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 138
./Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 138
./Platform.Data.Doublets/Unicode/UnicodeMap.cs, 139
./Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs, 141
./Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 142
./Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs, 142
./Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 143