

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(ICollection<TLink> restrictions)
19         {
20             var linkIndex = restrictions[Constants.IndexPart];
21             // Use Facade (the last decorator) to ensure recursion working correctly
22             Facade.DeleteAllUsages(linkIndex);
23             Links.Delete(linkIndex);
24         }
25     }
26 }

```

./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));
        ↪ return false;
127          }
128          if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
        ↪ target))
129          {
130              //throw new InvalidOperationException(string.Format("Невозможно продолжить
        ↪ путь через элемент пути {0}", next));
131

```



```

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 GetByKeyes<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 EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
    ↳ reference, string argumentName)
302 {
303     if (links.Constants.IsInnerReference(reference) && !links.Exists(reference))
304     {
305         throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
306     }
307 }
308
309 [MethodImpl(MethodImplOptions.AggressiveInlining)]
310 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
    ↳ IList<TLink> restrictions, string argumentName)
311 {
312     for (int i = 0; i < restrictions.Count; i++)
313     {
314         links.EnsureInnerReferenceExists(restrictions[i], argumentName);
315     }
316 }
317
318 [MethodImpl(MethodImplOptions.AggressiveInlining)]
319 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
    ↳ restrictions)
320 {
321     for (int i = 0; i < restrictions.Count; i++)

```

```

322     {
323         links.EnsureLinkIsAnyOrExists(restrictions[i], nameof(restrictions));
324     }
325 }
326
327 [MethodImpl(MethodImplOptions.AggressiveInlining)]
328 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
329     ↪ string argumentName)
330 {
331     var equalityComparer = EqualityComparer<TLink>.Default;
332     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
333     {
334         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
335     }
336 }
337
338 [MethodImpl(MethodImplOptions.AggressiveInlining)]
339 public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
340     ↪ link, string argumentName)
341 {
342     var equalityComparer = EqualityComparer<TLink>.Default;
343     if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
344     {
345         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
346     }
347 }
348
349 /// <param name="links">Хранилище связей.</param>
350 [MethodImpl(MethodImplOptions.AggressiveInlining)]
351 public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
352     ↪ TLink target)
353 {
354     if (links.Exists(source, target))
355     {
356         throw new LinkWithSameValueAlreadyExistsException();
357     }
358 }
359
360 /// <param name="links">Хранилище связей.</param>
361 public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
362 {
363     if (links.HasUsages(link))
364     {
365         throw new ArgumentLinkHasDependenciesException<TLink>(link);
366     }
367 }
368
369 /// <param name="links">Хранилище связей.</param>
370 public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
371     ↪ addresses) => links.EnsureCreated(links.Create, addresses);
372
373 /// <param name="links">Хранилище связей.</param>
374 public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
375     ↪ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
376
377 /// <param name="links">Хранилище связей.</param>
378 public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
379     ↪ params TLink[] addresses)
380 {
381     var constants = links.Constants;
382
383     var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
384     ↪ !links.Exists(x)));
385     if (nonExistentAddresses.Count > 0)
386     {
387         var max = nonExistentAddresses.Max();
388         max = (Integer<TLink>)System.Math.Min((ulong)(Integer<TLink>)max,
389     ↪ (ulong)(Integer<TLink>)constants.PossibleInnerReferencesRange.Maximum);
390         var createdLinks = new List<TLink>();
391         var equalityComparer = EqualityComparer<TLink>.Default;
392         TLink createdLink = creator();
393         while (!equalityComparer.Equals(createdLink, max))
394         {
395             createdLinks.Add(createdLink);
396         }
397         for (var i = 0; i < createdLinks.Count; i++)
398         {
399             if (!nonExistentAddresses.Contains(createdLinks[i]))

```

```

392         {
393             links.Delete(createdLinks[i]);
394         }
395     }
396 }
397
398 #endregion
399
400
401 /// <param name="links">Хранилище связей.</param>
402 public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
403 {
404     var constants = links.Constants;
405     var values = links.GetLink(link);
406     TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link,
407         ↪ constants.Any));
408     var equalityComparer = EqualityComparer<TLink>.Default;
409     if (equalityComparer.Equals(values[constants.SourcePart], link))
410     {
411         usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
412     }
413     TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
414         ↪ link));
415     if (equalityComparer.Equals(values[constants.TargetPart], link))
416     {
417         usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
418     }
419     return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
420 }
421
422 /// <param name="links">Хранилище связей.</param>
423 [MethodImpl(MethodImplOptions.AggressiveInlining)]
424 public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
425     ↪ Comparer<TLink>.Default.Compare(links.CountUsages(link), Integer<TLink>.Zero) > 0;
426
427 /// <param name="links">Хранилище связей.</param>
428 [MethodImpl(MethodImplOptions.AggressiveInlining)]
429 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
430     ↪ TLink target)
431 {
432     var constants = links.Constants;
433     var values = links.GetLink(link);
434     var equalityComparer = EqualityComparer<TLink>.Default;
435     return equalityComparer.Equals(values[constants.SourcePart], source) &&
436         ↪ equalityComparer.Equals(values[constants.TargetPart], target);
437 }
438
439 /// <summary>
440 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
441 /// </summary>
442 /// <param name="links">Хранилище связей.</param>
443 /// <param name="source">Индекс связи, которая является началом для искомой
444     ↪ связи.</param>
445 /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
446 /// <returns>Индекс искомой связи с указанными Source (началом) и Target
447     ↪ (концом).</returns>
448 [MethodImpl(MethodImplOptions.AggressiveInlining)]
449 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
450     ↪ target)
451 {
452     var constants = links.Constants;
453     var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break, default);
454     links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
455     return setter.Result;
456 }
457
458 /// <param name="links">Хранилище связей.</param>
459 [MethodImpl(MethodImplOptions.AggressiveInlining)]
460 public static TLink Create<TLink>(this ILinks<TLink> links) => links.Create(null);
461
462 /// <param name="links">Хранилище связей.</param>
463 [MethodImpl(MethodImplOptions.AggressiveInlining)]
464 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
465 {
466     var link = links.Create();
467     return links.Update(link, link, link);
468 }

```

```

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

465
466 /// <summary>
467 /// Обновляет связь с указанными началом (Source) и концом (Target)
468 /// на связь с указанными началом (NewSource) и концом (NewTarget).
469 /// </summary>
470 /// <param name="links">Хранилище связей.</param>
471 /// <param name="link">Индекс обновляемой связи.</param>
472 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    ↪ выполняется обновление.</param>
473 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    ↪ выполняется обновление.</param>
474 /// <returns>Индекс обновлённой связи.</returns>
475 [MethodImpl(MethodImplOptions.AggressiveInlining)]
476 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));

477
478 /// <summary>
479 /// Обновляет связь с указанными началом (Source) и концом (Target)
480 /// на связь с указанными началом (NewSource) и концом (NewTarget).
481 /// </summary>
482 /// <param name="links">Хранилище связей.</param>
483 /// <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>
484 /// <returns>Индекс обновлённой связи.</returns>
485 [MethodImpl(MethodImplOptions.AggressiveInlining)]
486 public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
487 {
488     if (restrictions.Length == 2)
489     {
490         return links.MergeAndDelete(restrictions[0], restrictions[1]);
491     }
492     if (restrictions.Length == 4)
493     {
494         return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
            ↪ restrictions[2], restrictions[3]);
495     }
496     else
497     {
498         return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
499     }
500 }

501
502 [MethodImpl(MethodImplOptions.AggressiveInlining)]
503 public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
    ↪ links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
504 {
505     var equalityComparer = EqualityComparer<TLink>.Default;
506     var constants = links.Constants;
507     var restrictionsIndex = restrictions[constants.IndexPart];
508     var substitutionIndex = substitution[constants.IndexPart];
509     if (equalityComparer.Equals(substitutionIndex, default))
510     {
511         substitutionIndex = restrictionsIndex;
512     }
513     var source = substitution[constants.SourcePart];
514     var target = substitution[constants.TargetPart];
515     source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
516     target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
517     return new Link<TLink>(substitutionIndex, source, target);
518 }

519
520 /// <summary>
521 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
    ↪ с указанными Source (началом) и Target (концом).
522 /// </summary>
523 /// <param name="links">Хранилище связей.</param>
524 /// <param name="source">Индекс связи, которая является началом на создаваемой
    ↪ связи.</param>
525 /// <param name="target">Индекс связи, которая является концом для создаваемой
    ↪ связи.</param>

```

```

526 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
527 [MethodImpl(MethodImplOptions.AggressiveInlining)]
528 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target)
529 {
530     var link = links.SearchOrDefault(source, target);
531     if (EqualityComparer<TLink>.Default.Equals(link, default))
532     {
533         link = links.CreateAndUpdate(source, target);
534     }
535     return link;
536 }
537
538 /// <summary>
539 /// Обновляет связь с указанными началом (Source) и концом (Target)
540 /// на связь с указанными началом (NewSource) и концом (NewTarget).
541 /// </summary>
542 /// <param name="links">Хранилище связей.</param>
543 /// <param name="source">Индекс связи, которая является началом обновляемой
    ↳ связи.</param>
544 /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
545 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    ↳ выполняется обновление.</param>
546 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    ↳ выполняется обновление.</param>
547 /// <returns>Индекс обновлённой связи.</returns>
548 [MethodImpl(MethodImplOptions.AggressiveInlining)]
549 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
    ↳ TLink target, TLink newSource, TLink newTarget)
550 {
551     var equalityComparer = EqualityComparer<TLink>.Default;
552     var link = links.SearchOrDefault(source, target);
553     if (equalityComparer.Equals(link, default))
554     {
555         return links.CreateAndUpdate(newSource, newTarget);
556     }
557     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
    ↳ target))
558     {
559         return link;
560     }
561     return links.Update(link, newSource, newTarget);
562 }
563
564 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
565 /// <param name="links">Хранилище связей.</param>
566 /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
567 /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
568 [MethodImpl(MethodImplOptions.AggressiveInlining)]
569 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target)
570 {
571     var link = links.SearchOrDefault(source, target);
572     if (!EqualityComparer<TLink>.Default.Equals(link, default))
573     {
574         links.Delete(link);
575         return link;
576     }
577     return default;
578 }
579
580 /// <summary>Удаляет несколько связей.</summary>
581 /// <param name="links">Хранилище связей.</param>
582 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
583 [MethodImpl(MethodImplOptions.AggressiveInlining)]
584 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
585 {
586     for (int i = 0; i < deletedLinks.Count; i++)
587     {
588         links.Delete(deletedLinks[i]);
589     }
590 }
591
592 /// <remarks>Before execution of this method ensure that deleted link is detached (all
    ↳ values - source and target are reset to null) or it might enter into infinite
    ↳ recursion.</remarks>
593 public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
594 {

```

```

595     var anyConstant = links.Constants.Any;
596     var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
597     links.DeleteByQuery(usagesAsSourceQuery);
598     var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
599     links.DeleteByQuery(usagesAsTargetQuery);
600 }
601
602 public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
603 {
604     var count = (Integer<TLink>)links.Count(query);
605     if (count > 0)
606     {
607         var queryResult = new TLink[count];
608         var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
        ↪ links.Constants.Continue);
609         links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
610         for (var i = (long)count - 1; i >= 0; i--)
611         {
612             links.Delete(queryResult[i]);
613         }
614     }
615 }
616
617 // TODO: Move to Platform.Data
618 public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
619 {
620     var nullConstant = links.Constants.Null;
621     var equalityComparer = EqualityComparer<TLink>.Default;
622     var link = links.GetLink(linkIndex);
623     for (int i = 1; i < link.Count; i++)
624     {
625         if (!equalityComparer.Equals(link[i], nullConstant))
626         {
627             return false;
628         }
629     }
630     return true;
631 }
632
633 // TODO: Create a universal version of this method in Platform.Data (with using of for
634 ↪ loop)
635 public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
636 {
637     var nullConstant = links.Constants.Null;
638     var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
639     links.Update(updateRequest);
640 }
641
642 // TODO: Create a universal version of this method in Platform.Data (with using of for
643 ↪ loop)
644 public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
645 {
646     if (!links.AreValuesReset(linkIndex))
647     {
648         links.ResetValues(linkIndex);
649     }
650 }
651
652 /// <summary>
653 /// Merging two usages graphs, all children of old link moved to be children of new link
654 ↪ or deleted.
655 /// </summary>
656 public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
657 ↪ TLink newLinkIndex)
658 {
659     var equalityComparer = EqualityComparer<TLink>.Default;
660     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
661     {
662         var constants = links.Constants;
663         var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
664             ↪ constants.Any);
665         long usagesAsSourceCount = (Integer<TLink>)links.Count(usagesAsSourceQuery);
666         var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
667             ↪ oldLinkIndex);
668         long usagesAsTargetCount = (Integer<TLink>)links.Count(usagesAsTargetQuery);
669         var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
670             ↪ usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
671         if (!isStandalonePoint)

```



```

665     {
666         var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
667         if (totalUsages > 0)
668         {
669             var usages = ArrayPool.Allocate<TLink>(totalUsages);
670             var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
671                 links.Constants.Continue);
672             var i = 0L;
673             if (usagesAsSourceCount > 0)
674             {
675                 links.Each(usagesFiller.AddFirstAndReturnConstant,
676                     ↪ usagesAsSourceQuery);
677                 for (; i < usagesAsSourceCount; i++)
678                 {
679                     var usage = usages[i];
680                     if (!equalityComparer.Equals(usage, oldLinkIndex))
681                     {
682                         links.Update(usage, newLinkIndex, links.GetTarget(usage));
683                     }
684                 }
685             }
686             if (usagesAsTargetCount > 0)
687             {
688                 links.Each(usagesFiller.AddFirstAndReturnConstant,
689                     ↪ usagesAsTargetQuery);
690                 for (; i < usages.Length; i++)
691                 {
692                     var usage = usages[i];
693                     if (!equalityComparer.Equals(usage, oldLinkIndex))
694                     {
695                         links.Update(usage, links.GetSource(usage), newLinkIndex);
696                     }
697                 }
698             }
699             ArrayPool.Free(usages);
700         }
701     }
702     return newLinkIndex;
703 }
704
705 /// <summary>
706 /// Replace one link with another (replaced link is deleted, children are updated or
707 /// ↪ deleted).
708 /// </summary>
709 [MethodImpl(MethodImplOptions.AggressiveInlining)]
710 public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
711     ↪ TLink newLinkIndex)
712 {
713     var equalityComparer = EqualityComparer<TLink>.Default;
714     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
715     {
716         links.MergeUsages(oldLinkIndex, newLinkIndex);
717         links.Delete(oldLinkIndex);
718     }
719     return newLinkIndex;
720 }
721
722 public static ILinks<TLink>
723     ↪ DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
724 {
725     links = new LinksCascadeUsagesResolver<TLink>(links);
726     links = new NonNullContentsLinkDeletionResolver<TLink>(links);
727     links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
728     return links;
729 }
730 }
731 }

```

./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 =
        ↳ EqualityComparer<TLink>.Default;
11
12     private readonly TLink _frequencyMarker;
13     private readonly TLink _unaryOne;
14     private readonly IIncrementer<TLink> _unaryNumberIncrementer;
15
16     public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
        ↳ IIncrementer<TLink> unaryNumberIncrementer)
        : base(links)
17     {
18
19         _frequencyMarker = frequencyMarker;
20         _unaryOne = unaryOne;
21         _unaryNumberIncrementer = unaryNumberIncrementer;
22     }
23
24     public TLink Increment(TLink frequency)
25     {
26         if (_equalityComparer.Equals(frequency, default))
27         {
28             return Links.GetOrCreate(_unaryOne, _frequencyMarker);
29         }
30         var source = Links.GetSource(frequency);
31         var incrementedSource = _unaryNumberIncrementer.Increment(source);
32         return Links.GetOrCreate(incrementedSource, _frequencyMarker);
33     }
34 }
35 }

```

./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 =
            ↳ EqualityComparer<TLink>.Default;
11
12         private readonly TLink _unaryOne;
13
14         public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
            ↳ _unaryOne = unaryOne;
15
16         public TLink Increment(TLink unaryNumber)
17         {
18             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
19             {
20                 return Links.GetOrCreate(_unaryOne, _unaryOne);
21             }
22             var source = Links.GetSource(unaryNumber);
23             var target = Links.GetTarget(unaryNumber);
24             if (_equalityComparer.Equals(source, target))
25             {
26                 return Links.GetOrCreate(unaryNumber, _unaryOne);
27             }
28             else
29             {
30                 return Links.GetOrCreate(source, Increment(target));
31             }
32         }
33     }
34 }

```

./Platform.Data.Doublets/ISynchronizedLinks.cs

```

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

```

./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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         public Link(TLink index, TLink source, TLink target)
62         {
63             Index = index;
64             Source = source;
65             Target = target;
66         }
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         private static void SetValues(ref Link<TLink> other, out TLink index, out TLink source,
70             ↪ out TLink target)
71         {
72             index = other.Index;
73             source = other.Source;
74             target = other.Target;
75         }
76
77         [MethodImpl(MethodImplOptions.AggressiveInlining)]
78         private static void SetValues(IList<TLink> values, out TLink index, out TLink source,
79             ↪ out TLink target)
```

```

74 {
75     switch (values.Count)
76     {
77         case 3:
78             index = values[0];
79             source = values[1];
80             target = values[2];
81             break;
82         case 2:
83             index = values[0];
84             source = values[1];
85             target = default;
86             break;
87         case 1:
88             index = values[0];
89             source = default;
90             target = default;
91             break;
92         default:
93             index = default;
94             source = default;
95             target = default;
96             break;
97     }
98 }
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 public override int GetHashCode() => (Index, Source, Target).GetHashCode();
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
105     && _equalityComparer.Equals(Source, _constants.Null)
106     && _equalityComparer.Equals(Target, _constants.Null);
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 public override bool Equals(object other) => other is Link<TLink> &&
110     ↪ Equals((Link<TLink>)other);
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
114     && _equalityComparer.Equals(Source, other.Source)
115     && _equalityComparer.Equals(Target, other.Target);
116
117 [MethodImpl(MethodImplOptions.AggressiveInlining)]
118 public static string ToString(TLink index, TLink source, TLink target) => $"{index}:
119     ↪ {source}->{target}";
120
121 [MethodImpl(MethodImplOptions.AggressiveInlining)]
122 public static string ToString(TLink source, TLink target) => $"{source}->{target}";
123
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 public static implicit operator TLink[](Link<TLink> link) => link.ToArray();
126
127 [MethodImpl(MethodImplOptions.AggressiveInlining)]
128 public static implicit operator Link<TLink>(TLink[] linkArray) => new
129     ↪ Link<TLink>(linkArray);
130
131 [MethodImpl(MethodImplOptions.AggressiveInlining)]
132 public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
133     ↪ ToString(Source, Target) : ToString(Index, Source, Target);
134
135 #region IList
136
137 public int Count => Length;
138
139 public bool IsReadOnly => true;
140
141 public TLink this[int index]
142 {
143     [MethodImpl(MethodImplOptions.AggressiveInlining)]
144     get
145     {
146         Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
147             ↪ nameof(index));
148         if (index == _constants.IndexPart)
149         {
150             return Index;
151         }
152         if (index == _constants.SourcePart)
153         {

```

```

149         return Source;
150     }
151     if (index == _constants.TargetPart)
152     {
153         return Target;
154     }
155     throw new NotSupportedException(); // Impossible path due to
        ↪ Ensure.ArgumentInRange
156 }
157 [MethodImpl(MethodImplOptions.AggressiveInlining)]
158 set => throw new NotSupportedException();
159 }
160
161 [MethodImpl(MethodImplOptions.AggressiveInlining)]
162 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
163
164 [MethodImpl(MethodImplOptions.AggressiveInlining)]
165 public IEnumerator<TLink> GetEnumerator()
166 {
167     yield return Index;
168     yield return Source;
169     yield return Target;
170 }
171
172 [MethodImpl(MethodImplOptions.AggressiveInlining)]
173 public void Add(TLink item) => throw new NotSupportedException();
174
175 [MethodImpl(MethodImplOptions.AggressiveInlining)]
176 public void Clear() => throw new NotSupportedException();
177
178 [MethodImpl(MethodImplOptions.AggressiveInlining)]
179 public bool Contains(TLink item) => IndexOf(item) >= 0;
180
181 [MethodImpl(MethodImplOptions.AggressiveInlining)]
182 public void CopyTo(TLink[] array, int arrayIndex)
183 {
184     Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
185     Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
        ↪ nameof(arrayIndex));
186     if (arrayIndex + Length > array.Length)
187     {
188         throw new InvalidOperationException();
189     }
190     array[arrayIndex++] = Index;
191     array[arrayIndex++] = Source;
192     array[arrayIndex] = Target;
193 }
194
195 [MethodImpl(MethodImplOptions.AggressiveInlining)]
196 public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
197
198 [MethodImpl(MethodImplOptions.AggressiveInlining)]
199 public int IndexOf(TLink item)
200 {
201     if (_equalityComparer.Equals(Index, item))
202     {
203         return _constants.IndexPart;
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         }
26     }
27 }
```

```

21     var one = Integer<TLink>.One;
22     var target = nullConstant;
23     for (int i = 0; !_equalityComparer.Equals(number, default) && i <
    ↪ NumericType<TLink>.BitsLength; i++)
24     {
25         if (_equalityComparer.Equals(Bit.And(number, one), one))
26         {
27             target = _equalityComparer.Equals(target, nullConstant)
28                 ? _powerOf2ToUnaryNumberConverter.Convert(i)
29                 : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
30         }
31         number = Bit.ShiftRight(number, 1);
32     }
33     return target;
34 }
35 }
36 }

```

./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>,
    ↪ IConverter<Doublet<TLink>, TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↪ EqualityComparer<TLink>.Default;
12
13         private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
14         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
15
16         public LinkToItsFrequencyNumberConveter(
17             ILinks<TLink> links,
18             IPropertyOperator<TLink, TLink> frequencyPropertyOperator,
19             IConverter<TLink> unaryNumberToAddressConverter)
20             : base(links)
21         {
22             _frequencyPropertyOperator = frequencyPropertyOperator;
23             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
24         }
25
26         public TLink Convert(Doublet<TLink> doublet)
27         {
28             var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
29             if (_equalityComparer.Equals(link, default))
30             {
31                 throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
32             }
33             var frequency = _frequencyPropertyOperator.Get(link);
34             if (_equalityComparer.Equals(frequency, default))
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>,
    ↪ IConverter<int, TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↪ EqualityComparer<TLink>.Default;
13

```

```

14     private readonly TLink[] _unaryNumberPowersOf2;
15
16     public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
17     {
18         _unaryNumberPowersOf2 = new TLink[64];
19         _unaryNumberPowersOf2[0] = one;
20     }
21
22     public TLink Convert(int power)
23     {
24         Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
25             ↳ - 1), nameof(power));
26         if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
27         {
28             return _unaryNumberPowersOf2[power];
29         }
30         var previousPowerOf2 = Convert(power - 1);
31         var powerOf2 = Links.GetOrCreate(previousPowerOf2, previousPowerOf2);
32         _unaryNumberPowersOf2[power] = powerOf2;
33         return powerOf2;
34     }
35 }

```

./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             {
31                 { _unaryOne, one }
32             };
33             var unary = _unaryOne;
34             var number = one;
35             for (var i = 1; i < 64; i++)
36             {
37                 unary = Links.GetOrCreate(unary, unary);
38                 number = Double(number);
39                 _unaryToUInt64.Add(unary, number);
40             }
41         }
42
43         public TLink Convert(TLink unaryNumber)
44         {
45             if (_equalityComparer.Equals(unaryNumber, default))
46             {
47                 return default;
48             }
49             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
50             {
51                 return Integer<TLink>.One;
52             }
53             var source = Links.GetSource(unaryNumber);
54             var target = Links.GetTarget(unaryNumber);
55             if (_equalityComparer.Equals(source, target))

```



```

54     {
55         return _unaryToUInt64[unaryNumber];
56     }
57     else
58     {
59         var result = _unaryToUInt64[source];
60         TLink lastValue;
61         while (!_unaryToUInt64.TryGetValue(target, out lastValue))
62         {
63             source = Links.GetSource(target);
64             result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
65             target = Links.GetTarget(target);
66         }
67         result = Arithmetic<TLink>.Add(result, lastValue);
68         return result;
69     }
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 private static TLink Double(TLink number) => (Integer<TLink>)((Integer<TLink>)number *
    ↳ 2UL);
74 }
75 }

```

./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>,
    ↳ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↳ EqualityComparer<TLink>.Default;
14
15         private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
16
17         public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
    ↳ TLink> powerOf2ToUnaryNumberConverter)
    ↳ : base(links)
18         {
19             _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
20             for (int i = 0; i < NumericType<TLink>.BitsLength; i++)
21             {
22                 _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
23             }
24         }
25
26         public TLink Convert(TLink sourceNumber)
27         {
28             var nullConstant = Links.Constants.Null;
29             var source = sourceNumber;
30             var target = nullConstant;
31             if (!_equalityComparer.Equals(source, nullConstant))
32             {
33                 while (true)
34                 {
35                     if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
36                     {
37                         SetBit(ref target, powerOf2Index);
38                         break;
39                     }
40                     else
41                     {
42                         powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
43                         SetBit(ref target, powerOf2Index);
44                         source = Links.GetTarget(source);
45                     }
46                 }
47             }
48             return target;
49         }
50     }
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

53         private static void SetBit(ref TLink target, int powerOf2Index) => target =
           ↪ Bit.Or(target, Bit.ShiftLeft(Integer<TLink>.One, powerOf2Index));
54     }
55 }

```

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

```

./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,
           ↪ TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪ EqualityComparer<TLink>.Default;
11
12         private readonly TLink _propertyMarker;
13         private readonly TLink _propertyValueMarker;
14
15         public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
           ↪ propertyValueMarker) : base(links)
16         {
17             _propertyMarker = propertyMarker;
18             _propertyValueMarker = propertyValueMarker;
19         }
20
21         public TLink Get(TLink link)
22         {
23             var property = Links.SearchOrDefault(link, _propertyMarker);
24             var container = GetContainer(property);
25             var value = GetValue(container);
26             return value;
27         }
28
29         private TLink GetContainer(TLink property)
30         {

```

```

31     var valueContainer = default(TLink);
32     if (_equalityComparer.Equals(property, default))
33     {
34         return valueContainer;
35     }
36     var constants = Links.Constants;
37     var countinueConstant = constants.Continue;
38     var breakConstant = constants.Break;
39     var anyConstant = constants.Any;
40     var query = new Link<TLink>(anyConstant, property, anyConstant);
41     Links.Each(candidate =>
42     {
43         var candidateTarget = Links.GetTarget(candidate);
44         var valueTarget = Links.GetTarget(candidateTarget);
45         if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
46         {
47             valueContainer = Links.GetIndex(candidate);
48             return breakConstant;
49         }
50         return countinueConstant;
51     }, query);
52     return valueContainer;
53 }
54
55 private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
56     ? default : Links.GetTarget(container);
57
58 public void Set(TLink link, TLink value)
59 {
60     var property = Links.GetOrCreate(link, _propertyMarker);
61     var container = GetContainer(property);
62     if (_equalityComparer.Equals(container, default))
63     {
64         Links.GetOrCreate(property, value);
65     }
66     else
67     {
68         Links.Update(container, property, value);
69     }
70 }
71 }

```

./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
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected abstract TLink GetTreeRoot();
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected abstract TLink GetBasePartValue(TLink link);
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

35     protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
    ↪     rootSource, TLink rootTarget);
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
    ↪     rootSource, TLink rootTarget);
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
    ↪     AsRef<LinksHeader<TLink>>(Header);
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
    ↪     AsRef<RawLink<TLink>>(Links + RawLink<TLink>.SizeInBytes * (Integer<TLink>)link);
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
48     {
49         ref var link = ref GetLinkReference(linkIndex);
50         return new Link<TLink>(linkIndex, link.Source, link.Target);
51     }
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
55     {
56         ref var firstLink = ref GetLinkReference(first);
57         ref var secondLink = ref GetLinkReference(second);
58         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
    ↪         secondLink.Source, secondLink.Target);
59     }
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
63     {
64         ref var firstLink = ref GetLinkReference(first);
65         ref var secondLink = ref GetLinkReference(second);
66         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↪         secondLink.Source, secondLink.Target);
67     }
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected virtual TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5,
    ↪     -5);
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     protected virtual void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
    ↪     Bit<TLink>.PartialWrite(storedValue, size, 5, -5);
74
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     protected virtual bool GetLeftIsChildValue(TLink value)
77     {
78         unchecked
79         {
80             //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 4, 1);
81             return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
82         }
83     }
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     protected virtual void SetLeftIsChildValue(ref TLink storedValue, bool value)
87     {
88         unchecked
89         {
90             var previousValue = storedValue;
91             var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 4,
    ↪             1);
92             storedValue = modified;
93         }
94     }
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     protected virtual bool GetRightIsChildValue(TLink value)
98     {
99         unchecked
100        {
101            //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 3, 1);
102            return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
103        }

```

```

104     }
105
106     [MethodImpl(MethodImplOptions.AggressiveInlining)]
107     protected virtual void SetRightIsChildValue(ref TLink storedValue, bool value)
108     {
109         unchecked
110         {
111             var previousValue = storedValue;
112             var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 3,
113                 ↪ 1);
114             storedValue = modified;
115         }
116     }
117
118     [MethodImpl(MethodImplOptions.AggressiveInlining)]
119     protected virtual sbyte GetBalanceValue(TLink storedValue)
120     {
121         unchecked
122         {
123             var value = (int)(Integer<TLink>)Bit<TLink>.PartialRead(storedValue, 0, 3);
124             value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
125                 ↪ end of sbyte
126             return (sbyte)value;
127         }
128     }
129
130     [MethodImpl(MethodImplOptions.AggressiveInlining)]
131     protected virtual void SetBalanceValue(ref TLink storedValue, sbyte value)
132     {
133         unchecked
134         {
135             var packagedValue = (TLink)(Integer<TLink>)((byte)value >> 5 & 4 | value & 3);
136             var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
137             storedValue = modified;
138         }
139     }
140
141     public TLink this[TLink index]
142     {
143         get
144         {
145             var root = GetTreeRoot();
146             if (GreaterOrEqualThan(index, GetSize(root)))
147             {
148                 return Zero;
149             }
150             while (!EqualToZero(root))
151             {
152                 var left = GetLeftOrDefault(root);
153                 var leftSize = GetSizeOrZero(left);
154                 if (LessThan(index, leftSize))
155                 {
156                     root = left;
157                     continue;
158                 }
159                 if (IsEquals(index, leftSize))
160                 {
161                     return root;
162                 }
163                 root = GetRightOrDefault(root);
164                 index = Subtract(index, Increment(leftSize));
165             }
166             return Zero; // TODO: Impossible situation exception (only if tree structure
167                 ↪ broken)
168         }
169     }
170
171     /// <summary>
172     /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
173     /// ↪ (концом).
174     /// </summary>
175     /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
176     /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
177     /// <returns>Индекс искомой связи.</returns>
178     public TLink Search(TLink source, TLink target)
179     {
180         var root = GetTreeRoot();
181         while (!EqualToZero(root))
182         {

```

```

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     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
233 }
234
235 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
236 {
237     var root = GetTreeRoot();
238     if (EqualToZero(root))
239     {
240         return Continue;
241     }
242     TLink first = Zero, current = root;
243     while (!EqualToZero(current))
244     {
245         var @base = GetBasePartValue(current);
246         if (GreaterOrEqualThan(@base, link))
247         {
248             if (IsEquals(@base, link))
249             {
250                 first = current;
251             }
252             current = GetLeftOrDefault(current);
253         }
254         else
255

```

```

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;
18         protected readonly byte* Links;
19         protected readonly byte* Header;
20
21         public LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
22             ↳ byte* header)
23         {
24             Links = links;
25             Header = header;
26             Break = constants.Break;
27             Continue = constants.Continue;
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected abstract TLink GetTreeRoot();
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected abstract TLink GetBasePartValue(TLink link);
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
38             ↳ rootSource, TLink rootTarget);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
42             ↳ rootSource, TLink rootTarget);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     }
46 }

```

```

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

```



```

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

```

```

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

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
6 {
7     public unsafe class LinksSourcesAvlBalancedTreeMethods<TLink> :
8     ↪ LinksAvlBalancedTreeMethodsBase<TLink>
9     {
10         public LinksSourcesAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
11         ↪ byte* header) : base(constants, links, header) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected unsafe override ref TLink GetLeftReference(TLink node) => ref
15         ↪ GetLinkReference(node).LeftAsSource;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected unsafe override ref TLink GetRightReference(TLink node) => ref
19         ↪ GetLinkReference(node).RightAsSource;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override void SetLeft(TLink node, TLink left) =>
29         ↪ GetLinkReference(node).LeftAsSource = left;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetRight(TLink node, TLink right) =>
33         ↪ GetLinkReference(node).RightAsSource = right;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override TLink GetSize(TLink node) =>
37         ↪ GetSizeValue(GetLinkReference(node).SizeAsSource);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
41         ↪ GetLinkReference(node).SizeAsSource, size);
42
43     }
44 }

```

```

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> :
    ↳ LinksSizeBalancedTreeMethodsBase<TLink>
8     {
9         public LinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
    ↳ byte* header) : base(constants, links, header) { }
10
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         protected unsafe override ref TLink GetLeftReference(TLink node) => ref
    ↳ GetLinkReference(node).LeftAsSource;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected unsafe override ref TLink GetRightReference(TLink node) => ref
    ↳ GetLinkReference(node).RightAsSource;
16
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) =>
25         ↪ GetLinkReference(node).LeftAsSource = left;
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     protected override void SetRight(TLink node, TLink right) =>
29         ↪ GetLinkReference(node).RightAsSource = right;
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsSource;
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected override void SetSize(TLink node, TLink size) =>
36         ↪ GetLinkReference(node).SizeAsSource = size;
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsSource;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
46         ↪ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
47         ↪ IsEquals(firstSource, secondSource) && LessThan(firstTarget, secondTarget);
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
51         ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
52         ↪ IsEquals(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget);
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected override void ClearNode(TLink node)
56     {
57         ref var link = ref GetLinkReference(node);
58         link.LeftAsSource = Zero;
59         link.RightAsSource = Zero;
60         link.SizeAsSource = Zero;
61     }
62 }

```

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

```

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

```

```

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

```

./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)]

```

```

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

```

./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;

```

```

32     protected readonly long _memoryReservationStep;
33
34     protected ILinksTreeMethods<TLink> TargetsTreeMethods;
35     protected ILinksTreeMethods<TLink> SourcesTreeMethods;
36     // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
37     ↪     нужно использовать не список а дерево, так как так можно быстрее проверить на
38     ↪     наличие связи внутри
39     protected ILinksListMethods<TLink> UnusedLinksListMethods;
40
41     /// <summary>
42     /// Возвращает общее число связей находящихся в хранилище.
43     /// </summary>
44     protected virtual TLink Total
45     {
46         get
47         {
48             ref var header = ref GetHeaderReference();
49             return Subtract(header.AllocatedLinks, header.FreeLinks);
50         }
51     }
52
53     public virtual LinksConstants<TLink> Constants { get; }
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     public ResizableDirectMemoryLinksBase(IResizableDirectMemory memory, long
57     ↪     memoryReservationStep, LinksConstants<TLink> constants)
58     {
59         _memory = memory;
60         _memoryReservationStep = memoryReservationStep;
61         Constants = constants;
62     }
63
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     public ResizableDirectMemoryLinksBase(IResizableDirectMemory memory, long
66     ↪     memoryReservationStep) : this(memory, memoryReservationStep,
67     ↪     Default<LinksConstants<TLink>>.Instance) { }
68
69     protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
70     {
71         if (memory.ReservedCapacity < memoryReservationStep)
72         {
73             memory.ReservedCapacity = memoryReservationStep;
74         }
75         SetPointers(_memory);
76         ref var header = ref GetHeaderReference();
77         // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
78         _memory.UsedCapacity = ConvertToUInt64(header.AllocatedLinks) * LinkSizeInBytes +
79         ↪     LinkHeaderSizeInBytes;
80         // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
81         header.ReservedLinks = ConvertToAddress((_memory.ReservedCapacity -
82         ↪     LinkHeaderSizeInBytes) / LinkSizeInBytes);
83     }
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     public virtual TLink Count(ICollection<TLink> restrictions)
87     {
88         // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
89         if (restrictions.Count == 0)
90         {
91             return Total;
92         }
93         var constants = Constants;
94         var any = constants.Any;
95         var index = restrictions[constants.IndexPart];
96         if (restrictions.Count == 1)
97         {
98             if (AreEqual(index, any))
99             {
100                 return Total;
101             }
102             return Exists(index) ? GetOne() : GetZero();
103         }
104         if (restrictions.Count == 2)
105         {
106             var value = restrictions[1];
107             if (AreEqual(index, any))
108             {
109                 if (AreEqual(value, any))
110                 {
111                     return Total;
112                 }
113             }
114         }
115     }

```

```

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 if (restrictions.Count == 3)
128 {
129     var source = restrictions[constants.SourcePart];
130     var target = restrictions[constants.TargetPart];
131     if (AreEqual(index, any))
132     {
133         if (AreEqual(source, any) && AreEqual(target, any))
134         {
135             return Total;
136         }
137         else if (AreEqual(source, any))
138         {
139             return TargetsTreeMethods.CountUsages(target);
140         }
141         else if (AreEqual(target, any))
142         {
143             return SourcesTreeMethods.CountUsages(source);
144         }
145         else //if(source != Any && target != Any)
146         {
147             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
148             var link = SourcesTreeMethods.Search(source, target);
149             return AreEqual(link, constants.Null) ? GetZero() : GetOne();
150         }
151     }
152 }
153 else
154 {
155     if (!Exists(index))
156     {
157         return GetZero();
158     }
159     if (AreEqual(source, any) && AreEqual(target, any))
160     {
161         return GetOne();
162     }
163     ref var storedLinkValue = ref GetLinkReference(index);
164     if (!AreEqual(source, any) && !AreEqual(target, any))
165     {
166         if (AreEqual(storedLinkValue.Source, source) &&
167             ↪ AreEqual(storedLinkValue.Target, target))
168         {
169             return GetOne();
170         }
171         return GetZero();
172     }
173     var value = default(TLink);
174     if (AreEqual(source, any))
175     {
176         value = target;
177     }
178     if (AreEqual(target, any))
179     {
180         value = source;
181     }

```



```

179         if (AreEqual(storedLinkValue.Source, value) ||
180             ↪ AreEqual(storedLinkValue.Target, value))
181         {
182             return GetOne();
183         }
184         return GetZero();
185     }
186     throw new NotSupportedException("Другие размеры и способы ограничений не
187     ↪ поддерживаются.");
188 }
189 [MethodImpl(MethodImplOptions.AggressiveInlining)]
190 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
191 {
192     var constants = Constants;
193     var @break = constants.Break;
194     if (restrictions.Count == 0)
195     {
196         for (var link = GetOne(); LessOrEqualThan(link,
197             ↪ GetHeaderReference().AllocatedLinks); link = Increment(link))
198         {
199             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
200             {
201                 return @break;
202             }
203         }
204         return @break;
205     }
206     var @continue = constants.Continue;
207     var any = constants.Any;
208     var index = restrictions[constants.IndexPart];
209     if (restrictions.Count == 1)
210     {
211         if (AreEqual(index, any))
212         {
213             return Each(handler, GetEmptyList());
214         }
215         if (!Exists(index))
216         {
217             return @continue;
218         }
219         return handler(GetLinkStruct(index));
220     }
221     if (restrictions.Count == 2)
222     {
223         var value = restrictions[1];
224         if (AreEqual(index, any))
225         {
226             if (AreEqual(value, any))
227             {
228                 return Each(handler, GetEmptyList());
229             }
230             if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
231             {
232                 return @break;
233             }
234             return Each(handler, new Link<TLink>(index, any, value));
235         }
236         else
237         {
238             if (!Exists(index))
239             {
240                 return @continue;
241             }
242             if (AreEqual(value, any))
243             {
244                 return handler(GetLinkStruct(index));
245             }
246             ref var storedLinkValue = ref GetLinkReference(index);
247             if (AreEqual(storedLinkValue.Source, value) ||
248                 AreEqual(storedLinkValue.Target, value))
249             {
250                 return handler(GetLinkStruct(index));
251             }
252             return @continue;
253         }
254     }
255 }

```

```

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 :
                ↪ handler(GetLinkStruct(link));
276         }
277     }
278     else
279     {
280         if (!Exists(index))
281         {
282             return @continue;
283         }
284         if (AreEqual(source, any) && AreEqual(target, any))
285         {
286             return handler(GetLinkStruct(index));
287         }
288         ref var storedLinkValue = ref GetLinkReference(index);
289         if (!AreEqual(source, any) && !AreEqual(target, any))
290         {
291             if (AreEqual(storedLinkValue.Source, source) &&
292                 AreEqual(storedLinkValue.Target, target))
293             {
294                 return handler(GetLinkStruct(index));
295             }
296             return @continue;
297         }
298         var value = default(TLink);
299         if (AreEqual(source, any))
300         {
301             value = target;
302         }
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
332     ↪ предварительно заполнено нулями
333     if (!AreEqual(link.Source, @null))
334     {
335         SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
336     }
337     if (!AreEqual(link.Target, @null))
338     {
339         TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
340     }
341     link.Source = substitution[constants.SourcePart];
342     link.Target = substitution[constants.TargetPart];
343     if (!AreEqual(link.Source, @null))
344     {
345         SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
346     }
347     if (!AreEqual(link.Target, @null))
348     {
349         TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
350     }
351     return linkIndex;
352 }
353
354 /// <remarks>
355 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
356 ↪ пространство
357 /// </remarks>
358 public virtual TLink Create(IList<TLink> restrictions)
359 {
360     ref var header = ref GetHeaderReference();
361     var freeLink = header.FirstFreeLink;
362     if (!AreEqual(freeLink, Constants.Null))
363     {
364         UnusedLinksListMethods.Detach(freeLink);
365     }
366     else
367     {
368         var maximumPossibleInnerReference =
369             ↪ Constants.PossibleInnerReferencesRange.Maximum;
370         if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
371         {
372             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
373         }
374         if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
375         {
376             _memory.ReservedCapacity += _memory.ReservationStep;
377             SetPointers(_memory);
378             header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
379                 ↪ LinkSizeInBytes);
380         }
381         header.AllocatedLinks = Increment(header.AllocatedLinks);
382         _memory.UsedCapacity += LinkSizeInBytes;
383         freeLink = header.AllocatedLinks;
384     }
385     return freeLink;
386 }
387
388 [MethodImpl(MethodImplOptions.AggressiveInlining)]
389 public virtual void Delete(IList<TLink> restrictions)
390 {
391     ref var header = ref GetHeaderReference();
392     var link = restrictions[Constants.IndexPart];
393     if (LessThan(link, header.AllocatedLinks)
394     {
395         UnusedLinksListMethods.AttachAsFirst(link);
396     }
397     else if (AreEqual(link, header.AllocatedLinks))
398     {
399         header.AllocatedLinks = Decrement(header.AllocatedLinks);
400         _memory.UsedCapacity -= LinkSizeInBytes;
401         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
402         ↪ пока не дойдём до первой существующей связи
403         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
404         while (GreaterThan(header.AllocatedLinks, GetZero()) &&
405             ↪ IsUnusedLink(header.AllocatedLinks))
406         {
407             UnusedLinksListMethods.Detach(header.AllocatedLinks);
408         }
409     }
410 }

```

```

402         header.AllocatedLinks = Decrement(header.AllocatedLinks);
403         _memory.UsedCapacity -= LinkSizeInBytes;
404     }
405 }
406
407 [MethodImpl(MethodImplOptions.AggressiveInlining)]
408 public IList<TLink> GetLinkStruct(TLink linkIndex)
409 {
410     ref var link = ref GetLinkReference(linkIndex);
411     return new Link<TLink>(linkIndex, link.Source, link.Target);
412 }
413
414 /// <remarks>
415 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
416 ↪ адрес реально поменялся
417 ///
418 /// Указатель this.links может быть в том же месте,
419 /// так как 0-я связь не используется и имеет такой же размер как Header,
420 /// поэтому header размещается в том же месте, что и 0-я связь
421 /// </remarks>
422 [MethodImpl(MethodImplOptions.AggressiveInlining)]
423 protected abstract void SetPointers(IResizableDirectMemory memory);
424
425 [MethodImpl(MethodImplOptions.AggressiveInlining)]
426 protected virtual void ResetPointers()
427 {
428     SourcesTreeMethods = null;
429     TargetsTreeMethods = null;
430     UnusedLinksListMethods = null;
431 }
432
433 [MethodImpl(MethodImplOptions.AggressiveInlining)]
434 protected abstract ref LinksHeader<TLink> GetHeaderReference();
435
436 [MethodImpl(MethodImplOptions.AggressiveInlining)]
437 protected abstract ref RawLink<TLink> GetLinkReference(TLink linkIndex);
438
439 [MethodImpl(MethodImplOptions.AggressiveInlining)]
440 protected virtual bool Exists(TLink link)
441     => GreaterOrEqualThan(link, Constants.PossibleInnerReferencesRange.Minimum)
442     && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
443     && !IsUnusedLink(link);
444
445 [MethodImpl(MethodImplOptions.AggressiveInlining)]
446 protected virtual bool IsUnusedLink(TLink linkIndex)
447 {
448     if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
449     ↪ is not needed
450     {
451         ref var link = ref GetLinkReference(linkIndex);
452         return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
453     }
454     else
455     {
456         return true;
457     }
458 }
459
460 [MethodImpl(MethodImplOptions.AggressiveInlining)]
461 protected virtual TLink GetOne() => Integer<TLink>.One;
462
463 [MethodImpl(MethodImplOptions.AggressiveInlining)]
464 protected virtual TLink GetZero() => Integer<TLink>.Zero;
465
466 [MethodImpl(MethodImplOptions.AggressiveInlining)]
467 protected virtual bool AreEqual(TLink first, TLink second) =>
468     ↪ EqualityComparer.Equals(first, second);
469
470 [MethodImpl(MethodImplOptions.AggressiveInlining)]
471 protected virtual bool LessThan(TLink first, TLink second) => Comparer.Compare(first,
472     ↪ second) < 0;
473
474 [MethodImpl(MethodImplOptions.AggressiveInlining)]
475 protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
476     ↪ Comparer.Compare(first, second) <= 0;
477
478 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

475     protected virtual bool GreaterThan(TLink first, TLink second) => Comparer.Compare(first,
476         ↪ second) > 0;
477
478     [MethodImpl(MethodImplOptions.AggressiveInlining)]
479     protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
480         ↪ Comparer.Compare(first, second) >= 0;
481
482     [MethodImpl(MethodImplOptions.AggressiveInlining)]
483     protected virtual long ConvertToUInt64(TLink value) => (Integer<TLink>)value;
484
485     [MethodImpl(MethodImplOptions.AggressiveInlining)]
486     protected virtual TLink ConvertToAddress(long value) => (Integer<TLink>)value;
487
488     [MethodImpl(MethodImplOptions.AggressiveInlining)]
489     protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
490         ↪ second);
491
492     [MethodImpl(MethodImplOptions.AggressiveInlining)]
493     protected virtual TLink Subtract(TLink first, TLink second) =>
494         ↪ Arithmetic<TLink>.Subtract(first, second);
495
496     [MethodImpl(MethodImplOptions.AggressiveInlining)]
497     protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
498
499     [MethodImpl(MethodImplOptions.AggressiveInlining)]
500     protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
501
502     [MethodImpl(MethodImplOptions.AggressiveInlining)]
503     protected virtual IList<TLink> GetEmptyList() => ArrayPool<TLink>.Empty;
504
505     #region Disposable
506
507     protected override bool AllowMultipleDisposeCalls => true;
508
509     protected override void Dispose(bool manual, bool wasDisposed)
510     {
511         if (!wasDisposed)
512         {
513             ResetPointers();
514             _memory.DisposeIfPossible();
515         }
516     }
517
518     #endregion
519 }

```

./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)]

```

```

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

```

./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     [MethodImpl(MethodImplOptions.AggressiveInlining)]
15     public UnusedLinksListMethods(byte* links, byte* header)
16     {
17         _links = links;
18         _header = header;
19     }
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
23     ↪ AsRef<LinksHeader<TLink>>(_header);
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
27     ↪ AsRef<RawLink<TLink>>(_links + RawLink<TLink>.SizeInBytes * (Integer<TLink>)link);
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override TLink GetPrevious(TLink element) => GetLinkReference(element).Source;
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override TLink GetNext(TLink element) => GetLinkReference(element).Target;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override TLink GetSize() => GetHeaderReference().FreeLinks;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
46     ↪ element;
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
50     ↪ element;
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected override void SetPrevious(TLink element, TLink previous) =>
54     ↪ GetLinkReference(element).Source = previous;
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected override void SetNext(TLink element, TLink next) =>
58     ↪ GetLinkReference(element).Target = next;
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
62 }

```

./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
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override ulong GetZero() => OUL;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override bool EqualToZero(ulong value) => value == OUL;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override bool IsEquals(ulong first, ulong second) => first == second;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override bool GreaterThanZero(ulong value) => value > OUL;
34     }
35 }
```



```

32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 protected override bool GreaterThan(ulong first, ulong second) => first > second;
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
37
38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
40 ↪ always true for ulong
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
44 ↪ always >= 0 for ulong
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
48
49 [MethodImpl(MethodImplOptions.AggressiveInlining)]
50 protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
51 ↪ for ulong
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected override bool LessThan(ulong first, ulong second) => first < second;
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 protected override ulong Increment(ulong value) => ++value;
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected override ulong Decrement(ulong value) => --value;
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override ulong Add(ulong first, ulong second) => first + second;
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override ulong Subtract(ulong first, ulong second) => first - second;
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
70 {
71     ref var firstLink = ref Links[first];
72     ref var secondLink = ref Links[second];
73     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
74 ↪ secondLink.Source, secondLink.Target);
75 }
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
79 {
80     ref var firstLink = ref Links[first];
81     ref var secondLink = ref Links[second];
82     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
83 ↪ secondLink.Source, secondLink.Target);
84 }
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 protected override ulong GetSizeValue(ulong value) => unchecked((value & 4294967264UL)
88 ↪ >> 5);
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
92 ↪ unchecked(storedValue & 31UL | (size & 134217727UL) << 5);
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 protected override bool GetLeftIsChildValue(ulong value) => unchecked((value & 16UL) >>
96 ↪ 4 == 1UL);
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
100 ↪ storedValue = unchecked(storedValue & 4294967279UL | (As<bool, byte>(ref value) &
101 ↪ 1UL) << 4);
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 protected override bool GetRightIsChildValue(ulong value) => unchecked((value & 8UL) >>
105 ↪ 3 == 1UL);
106
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

98     protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>
99         ↪ storedValue = unchecked(storedValue & 4294967287UL | (As<bool, byte>(ref value) &
100         ↪ 1UL) << 3);
101
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
107     [MethodImpl(MethodImplOptions.AggressiveInlining)]
108     protected override void SetBalanceValue(ref ulong storedValue, sbyte value) =>
109     ↪ storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
110     ↪ value & 3) & 7UL);
111
112     [MethodImpl(MethodImplOptions.AggressiveInlining)]
113     protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
114
115     [MethodImpl(MethodImplOptions.AggressiveInlining)]
116     protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
117 }
118 }

```

./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;
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;

```

```

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 FirstIsToLeftOfSecond(ulong first, ulong second)
67 {
68     ref var firstLink = ref Links[first];
69     ref var secondLink = ref Links[second];
70     return FirstIsToLeftOfSecond(firstLink.Source, firstLink.Target,
71         ↪ secondLink.Source, secondLink.Target);
72 }
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
76 {
77     ref var firstLink = ref Links[first];
78     ref var secondLink = ref Links[second];
79     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
80         ↪ secondLink.Source, secondLink.Target);
81 }
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
88 }

```

./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
6 {
7     public unsafe class UInt64LinksSourcesAvlBalancedTreeMethods :
8         ↪ UInt64LinksAvlBalancedTreeMethodsBase
9     {
10         public UInt64LinksSourcesAvlBalancedTreeMethods(LinksConstants<ulong> constants,
11             ↪ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
12             ↪ { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetLeftReference(ulong node) => ref
16             ↪ Links[node].LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20             ↪ Links[node].RightAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
30             ↪ left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
34             ↪ right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsSource);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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 :
    ↳ UInt64LinksSizeBalancedTreeMethodsBase
8      {
9          public UInt64LinksSourcesSizeBalancedTreeMethods(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) => Links[node].SizeAsSource;
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsSource =
        ↳ size;
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override ulong GetTreeRoot() => Header->FirstAsSource;
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
        ↳ ulong secondSource, ulong secondTarget)
43         => firstSource < secondSource || firstSource == secondSource && firstTarget <
        ↳ secondTarget;
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
        ↳ ulong secondSource, ulong secondTarget)
47         => firstSource > secondSource || firstSource == secondSource && firstTarget >
        ↳ secondTarget;
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override void ClearNode(ulong node)
51     {
52         ref var link = ref Links[node];
53         link.LeftAsSource = OUL;
54         link.RightAsSource = OUL;
55         link.SizeAsSource = OUL;
56     }
57 }
58 }

```

./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
6  {
7      public unsafe class UInt64LinksTargetsAvlBalancedTreeMethods :
8          ↳ UInt64LinksAvlBalancedTreeMethodsBase
9      {
10         public UInt64LinksTargetsAvlBalancedTreeMethods(LinksConstants<ulong> constants,
11             ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
12             ↳ { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetLeftReference(ulong node) => ref
16             ↳ Links[node].LeftAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20             ↳ Links[node].RightAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
30             ↳ left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
34             ↳ right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
41             ↳ size;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override ulong GetTreeRoot() => Header->FirstAsTarget;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
51             ↳ ulong secondSource, ulong secondTarget)
52             ↳ => firstSource < secondSource || firstSource == secondSource && firstTarget <
53                 ↳ secondTarget;
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
57             ↳ ulong secondSource, ulong secondTarget)
58             ↳ => firstSource > secondSource || firstSource == secondSource && firstTarget >
59                 ↳ secondTarget;
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         protected override void ClearNode(ulong node)
63         {
64             ref var link = ref Links[node];
65             link.LeftAsTarget = OUL;
66             link.RightAsTarget = OUL;
67             link.SizeAsTarget = OUL;
68         }
69     }
70 }

```

```

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

```

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

```

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

```

```

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

```

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

```

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

```

```

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

```



```

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

```

./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             {
46                 destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
47             }
48             if (length > loopedLength)
49             {
50                 destination[length / 2] = source[length - 1];
51             }
52         }
53     }

```

./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          _baseConverter = baseConverter;
71          _doubletFrequenciesCache = doubletFrequenciesCache;
72          if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
73          {
74              minFrequencyToCompress = Integer<TLink>.One;
75          }
76          _minFrequencyToCompress = minFrequencyToCompress;
77          _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
78          ResetMaxDoublet();
79      }
80
81      public override TLink Convert(IList<TLink> source) =>
82      ↪ _baseConverter.Convert(Compress(source));
83
84      /// <remarks>
85      /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
86      /// Faster version (doublets' frequencies dictionary is not recreated).
87      /// </remarks>
88      private IList<TLink> Compress(IList<TLink> sequence)
89      {
90          if (sequence.IsNullOrEmpty())
91          {
92              return null;
93          }
94          if (sequence.Count == 1)

```

```

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

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

```

```

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

```

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

```

```

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

```

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

```

```

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

```

./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>,
        ↪ ISequenceAppender<TLink>
11    {
12        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;
13
14        private readonly IStack<TLink> _stack;
15        private readonly ISequenceHeightProvider<TLink> _heightProvider;
16
17        public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
        ↪ ISequenceHeightProvider<TLink> heightProvider)
        : base(links)
18        {
19            _stack = stack;
20            _heightProvider = heightProvider;
21        }
22
23        public TLink Append(TLink sequence, TLink appendant)
24        {
25            var cursor = sequence;
26

```



```

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

```

./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 IList<TLink> _links;
22         private readonly IList<TLink> _sequences;
23         private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
24         private BitString _visited;
25
26         private class ItemEqualityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
27             ↪ IList<TLink>>>
28         {
29             private readonly IListEqualityComparer<TLink> _listComparer;
30             public ItemEqualityComparer() => _listComparer =
31             ↪ Default<IListEqualityComparer<TLink>>.Instance;
32         }
33     }
34 }

```

```

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

```

```

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

```

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

```

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

```

```

9 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
10 {
11     /// <remarks>
12     /// Can be used to operate with many CompressingConverters (to keep global frequencies data
13     /// ↪ between them).
14     /// TODO: Extract interface to implement frequencies storage inside Links storage
15     /// </remarks>
16     public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
17     {
18         private static readonly EqualityComparer<TLink> _equalityComparer =
19             ↪ EqualityComparer<TLink>.Default;
20         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
21
22         private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
23         private readonly ICounter<TLink, TLink> _frequencyCounter;
24
25         public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
26             : base(links)
27         {
28             _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
29                 ↪ DoubletComparer<TLink>.Default);
30             _frequencyCounter = frequencyCounter;
31
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
34             {
35                 var doublet = new Doublet<TLink>(source, target);
36                 return GetFrequency(ref doublet);
37             }
38
39             [MethodImpl(MethodImplOptions.AggressiveInlining)]
40             public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
41             {
42                 _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
43                 return data;
44             }
45
46             public void IncrementFrequencies(IList<TLink> sequence)
47             {
48                 for (var i = 1; i < sequence.Count; i++)
49                 {
50                     IncrementFrequency(sequence[i - 1], sequence[i]);
51                 }
52             }
53
54             [MethodImpl(MethodImplOptions.AggressiveInlining)]
55             public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
56             {
57                 var doublet = new Doublet<TLink>(source, target);
58                 return IncrementFrequency(ref doublet);
59             }
60
61             public void PrintFrequencies(IList<TLink> sequence)
62             {
63                 for (var i = 1; i < sequence.Count; i++)
64                 {
65                     PrintFrequency(sequence[i - 1], sequence[i]);
66                 }
67             }
68
69             public void PrintFrequency(TLink source, TLink target)
70             {
71                 var number = GetFrequency(source, target).Frequency;
72                 Console.WriteLine("{0},{1}) - {2}", source, target, number);
73             }
74
75             [MethodImpl(MethodImplOptions.AggressiveInlining)]
76             public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
77             {
78                 if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
79                 {
80                     data.IncrementFrequency();
81                 }
82                 else
83                 {
84                     var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
85                     data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
86                     if (!_equalityComparer.Equals(link, default))

```

```

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

```

./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                 // ...
35             }
36         }
37     }
38 }
```

```

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

```

./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 }
15
16 ./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs
17
18 using System.Collections.Generic;
19 using Platform.Interfaces;
20 using Platform.Numbers;
21
22 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
23
24 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
25 {
26     public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
27     {
28         private static readonly EqualityComparer<TLink> _equalityComparer =
29             ↳ EqualityComparer<TLink>.Default;
30         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
31
32         protected readonly ILinks<TLink> _links;
33         protected readonly TLink _symbol;
34         protected readonly HashSet<TLink> _visits;
35         protected TLink _total;
36
37         public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
38         {
39             _links = links;
40             _symbol = symbol;
41             _visits = new HashSet<TLink>();
42             _total = default;
43         }
44
45         public TLink Count()
46         {
47             if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
48             {
49                 return _total;
50             }
51             CountCore(_symbol);
52             return _total;
53         }
54
55         private void CountCore(TLink link)
56         {
57             var any = _links.Constants.Any;
58             if (_equalityComparer.Equals(_links.Count(any, link), default))
59             {
60                 CountSequenceSymbolFrequency(link);
61             }
62             else
63             {
64                 _links.Each(EachElementHandler, any, link);
65             }
66         }
67
68         protected virtual void CountSequenceSymbolFrequency(TLink link)
69         {
70             var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
71                 ↳ link, _symbol);
72             _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
73         }
74
75         private TLink EachElementHandler(IList<TLink> doublet)
76         {
77             var constants = _links.Constants;
78             var doubletIndex = doublet[constants.IndexPart];
79             if (_visits.Add(doubletIndex))
80             {
81                 CountCore(doubletIndex);
82             }
83             return constants.Continue;
84         }
85     }
86 }

```



```

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             return height;
51         }
52     }
53 }

```

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

```

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

```

```

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

```

./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             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(IList<TLink> sequence)
19         {
20             var indexed = true;
21             var i = sequence.Count;
22             while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
23                 ⇨ { }
24         }
25     }
26 }

```

```

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

```

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

```

```

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 =
10             ↳ EqualityComparer<TLink>.Default;
11
12         public SequenceIndex(ILinks<TLink> links) : base(links) { }
13
14         public virtual bool Add(IList<TLink> sequence)
15         {
16             var indexed = true;
17             var i = sequence.Count;
18             while (--i >= 1 && (indexed =
19                 ↳ !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
20                 ↳ default))) { }
21             for (; i >= 1; i--)
22             {
23                 Links.GetOrCreate(sequence[i - 1], sequence[i]);
24             }
25             return indexed;
26         }
27
28         public virtual bool MightContain(IList<TLink> sequence)
29         {
30             var indexed = true;
31             var i = sequence.Count;
32             while (--i >= 1 && (indexed =
33                 ↳ !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
34                 ↳ default))) { }
35             return indexed;
36         }
37     }
38 }

```

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

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

./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(ICollection<TElement> collection)
33     {
34         _list.Add(collection[0]);
35         return true;
36     }
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     public TReturnConstant AddAndReturnConstant(TElement element)
40     {
41         _list.Add(element);
42         return _returnConstant;
43     }
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> collection)
47     {
48         _list.Add(collection[0]);
49         return _returnConstant;
50     }
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     public TReturnConstant AddAllValuesAndReturnConstant(ICollection<TElement> collection)
54     {
55         for (int i = 1; i < collection.Count; i++)
56         {
57             _list.Add(collection[i]);
58         }
59         return _returnConstant;
60     }
61 }
62 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Runtime.CompilerServices;
5 using Platform.Collections;
6 using Platform.Collections.Lists;
7 using Platform.Threading.Synchronization;
8 using Platform.Singletons;
9 using LinkIndex = System.UInt64;
10 using Platform.Data.Doublets.Sequences.Walkers;
11 using Platform.Collections.Stacks;
12 using Platform.Collections.Arrays;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     /// <summary>
19     /// Представляет коллекцию последовательностей связей.
20     /// </summary>
21     /// <remarks>
22     /// Обязательно реализовать атомарность каждого публичного метода.
23     ///
24     /// TODO:

```

```

25  ///
26  /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
27  /// через естественную группировку по unicode типам, все whitespace вместе, все символы
    ↳ вместе, все числа вместе и т.п.
28  /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
    ↳ графа)
29  ///
30  /// х*у - найти все связи между, в последовательностях любой формы, если не стоит
    ↳ ограничитель на то, что является последовательностью, а что нет,
31  /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
    ↳ порядке.
32  ///
33  /// Рост последовательности слева и справа.
34  /// Поиск со звёздочкой.
35  /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
36  /// так же проблема может быть решена при реализации дистанционных триггеров.
37  /// Нужны ли уникальные указатели вообще?
38  /// Что если обращение к информации будет происходить через содержимое всегда?
39  ///
40  /// Писать тесты.
41  ///
42  ///
43  /// Можно убрать зависимость от конкретной реализации Links,
44  /// на зависимость от абстрактного элемента, который может быть представлен несколькими
    ↳ способами.
45  ///
46  /// Можно ли как-то сделать один общий интерфейс
47  ///
48  ///
49  /// Блокчейн и/или гит для распределённой записи транзакций.
50  ///
51  /// </remarks>
52  public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
    ↳ (после завершения реализации Sequences)
53  {
54      /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
    ↳ связей.</summary>
55      public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
56
57      public SequencesOptions<LinkIndex> Options { get; }
58      public SynchronizedLinks<LinkIndex> Links { get; }
59      private readonly ISynchronization _sync;
60
61      public LinksConstants<LinkIndex> Constants { get; }
62
63      public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
64      {
65          Links = links;
66          _sync = links.SyncRoot;
67          Options = options;
68          Options.ValidateOptions();
69          Options.InitOptions(Links);
70          Constants = Default<LinksConstants<LinkIndex>>.Instance;
71      }
72
73      public Sequences(SynchronizedLinks<LinkIndex> links)
74          : this(links, new SequencesOptions<LinkIndex>())
75      {
76      }
77
78      public bool IsSequence(LinkIndex sequence)
79      {
80          return _sync.ExecuteReadOperation(() =>
81          {
82              if (Options.UseSequenceMarker)
83              {
84                  return Options.MarkedSequenceMatcher.IsMatched(sequence);
85              }
86              return !Links.Unsync.IsPartialPoint(sequence);
87          });
88      }
89
90      [MethodImpl(MethodImplOptions.AggressiveInlining)]
91      private LinkIndex GetSequenceByElements(LinkIndex sequence)
92      {
93          if (Options.UseSequenceMarker)
94          {
95              return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
96          }

```

```

97         return sequence;
98     }
99
100 private LinkIndex GetSequenceElements(LinkIndex sequence)
101 {
102     if (Options.UseSequenceMarker)
103     {
104         var linkContents = new Link<ulong>(Links.GetLink(sequence));
105         if (linkContents.Source == Options.SequenceMarkerLink)
106         {
107             return linkContents.Target;
108         }
109         if (linkContents.Target == Options.SequenceMarkerLink)
110         {
111             return linkContents.Source;
112         }
113     }
114     return sequence;
115 }
116
117 #region Count
118
119 public LinkIndex Count(IList<LinkIndex> restrictions)
120 {
121     if (restrictions.IsNullOrEmpty())
122     {
123         return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
124     }
125     if (restrictions.Count == 1) // Первая связь это адрес
126     {
127         var sequenceIndex = restrictions[0];
128         if (sequenceIndex == Constants.Null)
129         {
130             return 0;
131         }
132         if (sequenceIndex == Constants.Any)
133         {
134             return Count(null);
135         }
136         if (Options.UseSequenceMarker)
137         {
138             return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
139         }
140         return Links.Exists(sequenceIndex) ? 1UL : 0;
141     }
142     throw new NotImplementedException();
143 }
144
145 private LinkIndex CountUsages(params LinkIndex[] restrictions)
146 {
147     if (restrictions.Length == 0)
148     {
149         return 0;
150     }
151     if (restrictions.Length == 1) // Первая связь это адрес
152     {
153         if (restrictions[0] == Constants.Null)
154         {
155             return 0;
156         }
157         if (Options.UseSequenceMarker)
158         {
159             var elementsLink = GetSequenceElements(restrictions[0]);
160             var sequenceLink = GetSequenceByElements(elementsLink);
161             if (sequenceLink != Constants.Null)
162             {
163                 return Links.Count(sequenceLink) + Links.Count(elementsLink) - 1;
164             }
165             return Links.Count(elementsLink);
166         }
167         return Links.Count(restrictions[0]);
168     }
169     throw new NotImplementedException();
170 }
171
172 #endregion
173
174 #region Create
175

```



```

176 public LinkIndex Create(IList<LinkIndex> restrictions)
177 {
178     return _sync.ExecuteWriteOperation(() =>
179     {
180         if (restrictions.IsNullOrEmpty())
181         {
182             return Constants.Null;
183         }
184         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
185         return CreateCore(restrictions);
186     });
187 }
188
189 private LinkIndex CreateCore(IList<LinkIndex> restrictions)
190 {
191     LinkIndex[] sequence = restrictions.ExtractValues();
192     if (Options.UseIndex)
193     {
194         Options.Index.Add(sequence);
195     }
196     var sequenceRoot = default(LinkIndex);
197     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
198     {
199         var matches = Each(restrictions);
200         if (matches.Count > 0)
201         {
202             sequenceRoot = matches[0];
203         }
204     }
205     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
206     {
207         return CompactCore(sequence);
208     }
209     if (sequenceRoot == default)
210     {
211         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
212     }
213     if (Options.UseSequenceMarker)
214     {
215         Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
216     }
217     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
218 }
219
220 #endregion
221
222 #region Each
223
224 public List<LinkIndex> Each(IList<LinkIndex> sequence)
225 {
226     var results = new List<LinkIndex>();
227     var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
228     Each(filler.AddFirstAndReturnConstant, sequence);
229     return results;
230 }
231
232 public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
↪ restrictions)
233 {
234     return _sync.ExecuteReadOperation(() =>
235     {
236         if (restrictions.IsNullOrEmpty())
237         {
238             return Constants.Continue;
239         }
240         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
241         if (restrictions.Count == 1)
242         {
243             var link = restrictions[0];
244             var any = Constants.Any;
245             if (link == any)
246             {
247                 if (Options.UseSequenceMarker)
248                 {
249                     return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
↪ Options.SequenceMarkerLink, any));
250                 }
251                 else

```

```

252         {
253             return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
254                 ↪ any));
255         }
256         var sequence =
257             ↪ Options.Walker.Walk(link).ToArray().ConvertToRestrictionsValues();
258         sequence[0] = link;
259         return handler(sequence);
260     }
261     else if (restrictions.Count == 2)
262     {
263         throw new NotImplementedException();
264     }
265     else if (restrictions.Count == 3)
266     {
267         return Links.Unsync.Each(handler, restrictions);
268     }
269     else
270     {
271         var sequence = restrictions.ExtractValues();
272         if (Options.UseIndex && !Options.Index.MightContain(sequence))
273         {
274             return Constants.Break;
275         }
276         return EachCore(handler, sequence);
277     }
278 }
279
280 private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
281     ↪ values)
282 {
283     var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
284     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
285     ↪ Id.
286     Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
287     ↪ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
288     ↪ matcher.HandleFullMatched;
289     //if (sequence.Length >= 2)
290     if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
291     {
292         return Constants.Break;
293     }
294     var last = values.Count - 2;
295     for (var i = 1; i < last; i++)
296     {
297         if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
298             ↪ Constants.Continue)
299         {
300             return Constants.Break;
301         }
302     }
303     if (values.Count >= 3)
304     {
305         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
306             ↪ != Constants.Continue)
307         {
308             return Constants.Break;
309         }
310     }
311     return Constants.Continue;
312 }
313
314 private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
315     ↪ left, LinkIndex right)
316 {
317     return Links.Unsync.Each(doublet =>
318     {
319         var doubletIndex = doublet[Constants.IndexPart];
320         if (StepRight(handler, doubletIndex, right) != Constants.Continue)
321         {
322             return Constants.Break;
323         }
324         if (left != doubletIndex)
325         {
326             return PartialStepRight(handler, doubletIndex, right);
327         }
328     });

```

```

321         return Constants.Continue;
322     }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
323 }
324
325 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));
326
327 private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ right, LinkIndex stepFrom)
328 {
329     var upStep = stepFrom;
330     var firstSource = Links.Unsync.GetTarget(upStep);
331     while (firstSource != right && firstSource != upStep)
332     {
333         upStep = firstSource;
334         firstSource = Links.Unsync.GetSource(upStep);
335     }
336     if (firstSource == right)
337     {
338         return handler(new LinkAddress<LinkIndex>(stepFrom));
339     }
340     return Constants.Continue;
341 }
342
343 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));
344
345 private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ left, LinkIndex stepFrom)
346 {
347     var upStep = stepFrom;
348     var firstTarget = Links.Unsync.GetSource(upStep);
349     while (firstTarget != left && firstTarget != upStep)
350     {
351         upStep = firstTarget;
352         firstTarget = Links.Unsync.GetTarget(upStep);
353     }
354     if (firstTarget == left)
355     {
356         return handler(new LinkAddress<LinkIndex>(stepFrom));
357     }
358     return Constants.Continue;
359 }
360
361 #endregion
362
363 #region Update
364
365 public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
366 {
367     var sequence = restrictions.ExtractValues();
368     var newSequence = substitution.ExtractValues();
369
370     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
371     {
372         return Constants.Null;
373     }
374     if (sequence.IsNullOrEmpty())
375     {
376         return Create(substitution);
377     }
378     if (newSequence.IsNullOrEmpty())
379     {
380         Delete(restrictions);
381         return Constants.Null;
382     }
383     return _sync.ExecuteWriteOperation(() =>
384     {
385         Links.EnsureEachLinkIsAnyOrExists(sequence);
386         Links.EnsureEachLinkExists(newSequence);
387         return UpdateCore(sequence, newSequence);
388     });
389 }
390
391 private LinkIndex UpdateCore(LinkIndex[] sequence, LinkIndex[] newSequence)

```

```

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

```

```

467 {
468     _sync.ExecuteWriteOperation(() =>
469     {
470         var sequence = restrictions.ExtractValues();
471         // TODO: Check all options only ones before loop execution
472         foreach (var linkToDelete in Each(sequence))
473         {
474             DeleteOneCore(linkToDelete);
475         }
476     });
477 }
478
479 private void DeleteOneCore(LinkIndex link)
480 {
481     if (Options.UseGarbageCollection)
482     {
483         var sequenceElements = GetSequenceElements(link);
484         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
485         var sequenceLink = GetSequenceByElements(sequenceElements);
486         if (Options.UseCascadeDelete || CountUsages(link) == 0)
487         {
488             if (sequenceLink != Constants.Null)
489             {
490                 Links.Unsync.Delete(sequenceLink);
491             }
492             Links.Unsync.Delete(link);
493         }
494         ClearGarbage(sequenceElementsContents.Source);
495         ClearGarbage(sequenceElementsContents.Target);
496     }
497     else
498     {
499         if (Options.UseSequenceMarker)
500         {
501             var sequenceElements = GetSequenceElements(link);
502             var sequenceLink = GetSequenceByElements(sequenceElements);
503             if (Options.UseCascadeDelete || CountUsages(link) == 0)
504             {
505                 if (sequenceLink != Constants.Null)
506                 {
507                     Links.Unsync.Delete(sequenceLink);
508                 }
509                 Links.Unsync.Delete(link);
510             }
511         }
512         else
513         {
514             if (Options.UseCascadeDelete || CountUsages(link) == 0)
515             {
516                 Links.Unsync.Delete(link);
517             }
518         }
519     }
520 }
521
522 #endregion
523
524 #region Compactification
525
526 /// <remarks>
527 /// bestVariant можно выбирать по максимальному числу использований,
528 /// но балансированный позволяет гарантировать уникальность (если есть возможность,
529 /// гарантировать его использование в других местах).
530 ///
531 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
532 /// </remarks>
533 public LinkIndex Compact(params LinkIndex[] sequence)
534 {
535     return _sync.ExecuteWriteOperation(() =>
536     {
537         if (sequence.IsNullOrEmpty())
538         {
539             return Constants.Null;
540         }
541         Links.EnsureEachLinkExists(sequence);
542         return CompactCore(sequence);
543     });
544 }

```

```

545 [MethodImpl(MethodImplOptions.AggressiveInlining)]
546 private LinkIndex CompactCore(params LinkIndex[] sequence) => UpdateCore(sequence,
547     ↪ sequence);
548
549 #endregion
550
551 #region Garbage Collection
552
553 /// <remarks>
554 /// TODO0: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
555 ↪ определить извне или в унаследованном классе
556 /// </remarks>
557 [MethodImpl(MethodImplOptions.AggressiveInlining)]
558 private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
559     ↪ !Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;
560
561 private void ClearGarbage(LinkIndex link)
562 {
563     if (IsGarbage(link))
564     {
565         var contents = new Link<ulong>(Links.GetLink(link));
566         Links.Unsync.Delete(link);
567         ClearGarbage(contents.Source);
568         ClearGarbage(contents.Target);
569     }
570 }
571
572 #endregion
573
574 #region Walkers
575
576 public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
577 {
578     return _sync.ExecuteReadOperation(() =>
579     {
580         var links = Links.Unsync;
581         foreach (var part in Options.Walker.Walk(sequence))
582         {
583             if (!handler(part))
584             {
585                 return false;
586             }
587         }
588         return true;
589     });
590 }
591
592 public class Matcher : RightSequenceWalker<LinkIndex>
593 {
594     private readonly Sequences _sequences;
595     private readonly IList<LinkIndex> _patternSequence;
596     private readonly HashSet<LinkIndex> _linksInSequence;
597     private readonly HashSet<LinkIndex> _results;
598     private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
599     private readonly HashSet<LinkIndex> _readAsElements;
600     private int _filterPosition;
601
602     public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
603         ↪ HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
604         ↪ HashSet<LinkIndex> readAsElements = null)
605         : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
606     {
607         _sequences = sequences;
608         _patternSequence = patternSequence;
609         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
610             ↪ Links.Constants.Any && x != ZeroOrMany));
611         _results = results;
612         _stopableHandler = stopableHandler;
613         _readAsElements = readAsElements;
614     }
615
616     protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
617         ↪ (_readAsElements != null && _readAsElements.Contains(link)) ||
618         ↪ _linksInSequence.Contains(link);
619
620     public bool FullMatch(LinkIndex sequenceToMatch)
621     {
622         _filterPosition = 0;
623         foreach (var part in Walk(sequenceToMatch))

```

```

617         {
618             if (!FullMatchCore(part))
619             {
620                 break;
621             }
622         }
623         return _filterPosition == _patternSequence.Count;
624     }
625
626     private bool FullMatchCore(LinkIndex element)
627     {
628         if (_filterPosition == _patternSequence.Count)
629         {
630             _filterPosition = -2; // Длиннее чем нужно
631             return false;
632         }
633         if (_patternSequence[_filterPosition] != Links.Constants.Any
634             && element != _patternSequence[_filterPosition])
635         {
636             _filterPosition = -1;
637             return false; // Начинается/Продолжается иначе
638         }
639         _filterPosition++;
640         return true;
641     }
642
643     public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
644     {
645         var sequenceToMatch = restrictions[Links.Constants.IndexPart];
646         if (FullMatch(sequenceToMatch))
647         {
648             _results.Add(sequenceToMatch);
649         }
650     }
651
652     public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
653     {
654         var sequenceToMatch = restrictions[Links.Constants.IndexPart];
655         if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
656         {
657             return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
658         }
659         return Links.Constants.Continue;
660     }
661
662     public LinkIndex HandleFullMatchedSequence(IList<LinkIndex> restrictions)
663     {
664         var sequenceToMatch = restrictions[Links.Constants.IndexPart];
665         var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
666         if (sequence != Links.Constants.Null && FullMatch(sequenceToMatch) &&
667             ↪ _results.Add(sequenceToMatch))
668         {
669             return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
670         }
671         return Links.Constants.Continue;
672     }
673
674     /// <remarks>
675     /// TODO: Add support for LinksConstants.Any
676     /// </remarks>
677     public bool PartialMatch(LinkIndex sequenceToMatch)
678     {
679         _filterPosition = -1;
680         foreach (var part in Walk(sequenceToMatch))
681         {
682             if (!PartialMatchCore(part))
683             {
684                 break;
685             }
686         }
687         return _filterPosition == _patternSequence.Count - 1;
688     }
689
690     private bool PartialMatchCore(LinkIndex element)
691     {
692         if (_filterPosition == (_patternSequence.Count - 1))
693         {
694             return false; // Нашлось
695         }

```

```

695         if (_filterPosition >= 0)
696         {
697             if (element == _patternSequence[_filterPosition + 1])
698             {
699                 _filterPosition++;
700             }
701             else
702             {
703                 _filterPosition = -1;
704             }
705         }
706         if (_filterPosition < 0)
707         {
708             if (element == _patternSequence[0])
709             {
710                 _filterPosition = 0;
711             }
712         }
713         return true; // Ищем дальше
714     }
715
716     public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
717     {
718         if (PartialMatch(sequenceToMatch))
719         {
720             _results.Add(sequenceToMatch);
721         }
722     }
723
724     public LinkIndex HandlePartialMatched(ICollection<LinkIndex> restrictions)
725     {
726         var sequenceToMatch = restrictions[Links.Constants.IndexPart];
727         if (PartialMatch(sequenceToMatch))
728         {
729             return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
730         }
731         return Links.Constants.Continue;
732     }
733
734     public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)
735     {
736         foreach (var sequenceToMatch in sequencesToMatch)
737         {
738             if (PartialMatch(sequenceToMatch))
739             {
740                 _results.Add(sequenceToMatch);
741             }
742         }
743     }
744
745     public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
746 ↪ sequencesToMatch)
747     {
748         foreach (var sequenceToMatch in sequencesToMatch)
749         {
750             if (PartialMatch(sequenceToMatch))
751             {
752                 _readAsElements.Add(sequenceToMatch);
753                 _results.Add(sequenceToMatch);
754             }
755         }
756     }
757
758     #endregion
759 }
760 }

```

./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.EnsureEachLinkExists(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())

```

```

86         {
87             return new List<ulong>();
88         }
89         Links.Unsync.EnsureEachLinkExists(sequence);
90         if (sequence.Length == 1)
91         {
92             return new List<ulong> { sequence[0] };
93         }
94         var results = new
95             ↪ List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
96         return CreateAllVariants1Core(sequence, results);
97     });
98 }
99 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
100 {
101     if (sequence.Length == 2)
102     {
103         var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);
104         if (link == Constants.Null)
105         {
106             throw new NotImplementedException("Creation cancellation is not
107                 ↪ implemented.");
108         }
109         results.Add(link);
110         return results;
111     }
112     var innerSequenceLength = sequence.Length - 1;
113     var innerSequence = new ulong[innerSequenceLength];
114     for (var li = 0; li < innerSequenceLength; li++)
115     {
116         var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
117         if (link == Constants.Null)
118         {
119             throw new NotImplementedException("Creation cancellation is not
120                 ↪ implemented.");
121         }
122         for (var isi = 0; isi < li; isi++)
123         {
124             innerSequence[isi] = sequence[isi];
125         }
126         innerSequence[li] = link;
127         for (var isi = li + 1; isi < innerSequenceLength; isi++)
128         {
129             innerSequence[isi] = sequence[isi + 1];
130         }
131         CreateAllVariants1Core(innerSequence, results);
132     }
133     return results;
134 }
135 #endregion
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 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
150 {
151     if (sequence.Length == 2)
152     {
153         Links.Unsync.Each(sequence[0], sequence[1], handler);
154     }
155     else
156     {
157         var innerSequenceLength = sequence.Length - 1;
158         for (var li = 0; li < innerSequenceLength; li++)
159         {
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             innerSequence[linkIndex] = doublet[Constants.IndexPart];
183             Each1(handler, innerSequence);
184             return Constants.Continue;
185         }, Constants.Any, left, right);
186     }
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.EnsureEachLinkIsAnyOrExists(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
242     {
243         EachPart(handler, sequence);
244     }
245 }

```

```

238     }
239 }
240 else if (sequence.Length == 2)
241 {
242     //_links.Each(sequence[0], sequence[1], handler);
243     //  o_|      x_o ...
244     // x_|      |___|
245     Links.Each(sequence[1], Constants.Any, doublet =>
246     {
247         var match = Links.SearchOrDefault(sequence[0], doublet);
248         if (match != Constants.Null)
249         {
250             handler(new LinkAddress<LinkIndex>(match));
251         }
252         return true;
253     });
254     // |_x      ... x_o
255     // |_o      |___|
256     Links.Each(Constants.Any, sequence[0], doublet =>
257     {
258         var match = Links.SearchOrDefault(doublet, sequence[1]);
259         if (match != 0)
260         {
261             handler(new LinkAddress<LinkIndex>(match));
262         }
263         return true;
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
↳ stepFrom)
298 {
299     var upStep = stepFrom;
300     var firstSource = Links.Unsync.GetTarget(upStep);
301     while (firstSource != right && firstSource != upStep)
302     {
303         upStep = firstSource;
304         firstSource = Links.Unsync.GetSource(upStep);
305     }
306     if (firstSource == right)
307     {
308         handler(new LinkAddress<LinkIndex>(stepFrom));
309     }
310 }
311
312 // TODO: Test
313 private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
314 {

```

```

315     Links.Unsync.Each(right, Constants.Any, doublet =>
316     {
317         StepLeft(handler, left, doublet);
318         if (right != doublet)
319         {
320             PartialStepLeft(handler, left, doublet);
321         }
322         return true;
323     });
324 }
325
326 private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
327 {
328     Links.Unsync.Each(Constants.Any, right, leftStep =>
329     {
330         TryStepLeftUp(handler, left, leftStep);
331         return true;
332     });
333 }
334
335 private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
336 {
337     var upStep = stepFrom;
338     var firstTarget = Links.Unsync.GetSource(upStep);
339     while (firstTarget != left && firstTarget != upStep)
340     {
341         upStep = firstTarget;
342         firstTarget = Links.Unsync.GetTarget(upStep);
343     }
344     if (firstTarget == left)
345     {
346         handler(new LinkAddress<LinkIndex>(stepFrom));
347     }
348 }
349
350 private bool StartsWith(ulong sequence, ulong link)
351 {
352     var upStep = sequence;
353     var firstSource = Links.Unsync.GetSource(upStep);
354     while (firstSource != link && firstSource != upStep)
355     {
356         upStep = firstSource;
357         firstSource = Links.Unsync.GetSource(upStep);
358     }
359     return firstSource == link;
360 }
361
362 private bool EndsWith(ulong sequence, ulong link)
363 {
364     var upStep = sequence;
365     var lastTarget = Links.Unsync.GetTarget(upStep);
366     while (lastTarget != link && lastTarget != upStep)
367     {
368         upStep = lastTarget;
369         lastTarget = Links.Unsync.GetTarget(upStep);
370     }
371     return lastTarget == link;
372 }
373
374 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
375 {
376     return _sync.ExecuteReadOperation(() =>
377     {
378         var results = new List<ulong>();
379         if (sequence.Length > 0)
380         {
381             Links.EnsureEachLinkExists(sequence);
382             var firstElement = sequence[0];
383             if (sequence.Length == 1)
384             {
385                 results.Add(firstElement);
386                 return results;
387             }
388             if (sequence.Length == 2)
389             {
390                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
391                 if (doublet != Constants.Null)
392                 {
393                     results.Add(doublet);

```

```

394     }
395     return results;
396 }
397 var linksInSequence = new HashSet<ulong>(sequence);
398 void handler(IList<LinkIndex> result)
399 {
400     var resultIndex = result[Links.Constants.IndexPart];
401     var filterPosition = 0;
402     StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
403     ↪ Links.Unsync.GetTarget,
404     ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
405     ↪ x =>
406     {
407         if (filterPosition == sequence.Length)
408         {
409             filterPosition = -2; // Длиннее чем нужно
410             return false;
411         }
412         if (x != sequence[filterPosition])
413         {
414             filterPosition = -1;
415             return false; // Начинается иначе
416         }
417         filterPosition++;
418         return true;
419     });
420     if (filterPosition == sequence.Length)
421     {
422         results.Add(resultIndex);
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.EnsureEachLinkExists(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 public const int MaxSequenceFormatSize = 200;
486
487 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
488     ↪ => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
489
490 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
491     ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
492     ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
493     ↪ elementToString, insertComma, knownElements));
494
495 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
496     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
497     ↪ LinkIndex[] knownElements)
498 {
499     var linksInSequence = new HashSet<ulong>(knownElements);
500     //var entered = new HashSet<ulong>();
501     var sb = new StringBuilder();
502     sb.Append('{');
503     if (links.Exists(sequenceLink))
504     {
505         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
506             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
507             ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
508         {
509             if (insertComma && sb.Length > 1)
510             {
511                 sb.Append(',');
512             }
513             //if (entered.Contains(element))
514             //if (
515             //    sb.Append('{');
516             //    elementToString(sb, element);
517             //    sb.Append('}');
518             //}
519             //else
520             elementToString(sb, element);
521             if (sb.Length < MaxSequenceFormatSize)
522             {
523                 return true;
524             }
525             sb.Append(insertComma ? ", ..." : "...");
526             return false;
527         });
528     }
529     sb.Append('}');
530     return sb.ToString();
531 }
532
533 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
534     ↪ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
535     ↪ knownElements);
536
537 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
538     ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
539     ↪ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
540     ↪ sequenceLink, elementToString, insertComma, knownElements));
541
542 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
543     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
544     ↪ LinkIndex[] knownElements)
545 {

```

```

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),
539             entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
540             {
541                 if (insertComma && sb.Length > 1)
542                 {
543                     sb.Append(',');
544                 }
545                 if (entered.Contains(element))
546                 {
547                     sb.Append('{');
548                     elementToString(sb, element);
549                     sb.Append('}');
550                 }
551                 else
552                 {
553                     elementToString(sb, element);
554                 }
555                 if (sb.Length < MaxSequenceFormatSize)
556                 {
557                     return true;
558                 }
559                 sb.Append(insertComma ? ", ..." : "...");
560                 return false;
561             });
562     }
563     sb.Append('}');
564     return sb.ToString();
565 }
566
567 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
568 {
569     return _sync.ExecuteReadOperation(() =>
570     {
571         if (sequence.Length > 0)
572         {
573             Links.EnsureEachLinkExists(sequence);
574             var results = new HashSet<ulong>();
575             for (var i = 0; i < sequence.Length; i++)
576             {
577                 AllUsagesCore(sequence[i], results);
578             }
579             var filteredResults = new List<ulong>();
580             var linksInSequence = new HashSet<ulong>(sequence);
581             foreach (var result in results)
582             {
583                 var filterPosition = -1;
584                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
585                     Links.Unsync.GetTarget,
586                     x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
587                     x =>
588                     {
589                         if (filterPosition == (sequence.Length - 1))
590                         {
591                             return false;
592                         }
593                     }
594                     if (filterPosition >= 0)
595                     {
596                         if (x == sequence[filterPosition + 1])
597                         {
598                             filterPosition++;
599                         }
600                         else
601                         {
602                             return false;
603                         }
604                     }
605                     if (filterPosition < 0)
606                     {
607                         if (x == sequence[0])
608                         {
609                             filterPosition = 0;
610                         }
611                     }
612                 }
613             }
614             filteredResults.AddRange(results);
615         }
616     });
617 }

```



```

606         }
607     }
608     return true;
609 });
610     if (filterPosition == (sequence.Length - 1))
611     {
612         filteredResults.Add(result);
613     }
614 }
615 return filteredResults;
616 }
617 return new List<ulong>();
618 });
619 }
620
621 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
622 {
623     return _sync.ExecuteReadOperation(() =>
624     {
625         if (sequence.Length > 0)
626         {
627             Links.EnsureEachLinkExists(sequence);
628             var results = new HashSet<ulong>();
629             for (var i = 0; i < sequence.Length; i++)
630             {
631                 AllUsagesCore(sequence[i], results);
632             }
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.EnsureEachLinkExists(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);

```

```

684 //          //for (var i = 0; i < sequence.Length; i++)
685 //          //    AllUsagesCore(sequence[i], results);
686 //
687 //          var filteredResults = new HashSet<ulong>();
688 //          var matcher = new Matcher(this, sequence, filteredResults, null);
689 //          matcher.AddAllPartialMatchedToResults(firstResults);
690 //          return filteredResults;
691 //      }
692 //
693 //      return new HashSet<ulong>();
694 //  });
695 //}
696
697
698 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
699 {
700     return _sync.ExecuteReadOperation(() =>
701     {
702         if (sequence.Length > 0)
703         {
704             Links.EnsureEachLinkIsAnyOrExists(sequence);
705             var firstResults = new HashSet<ulong>();
706             var lastResults = new HashSet<ulong>();
707             var first = sequence.First(x => x != Constants.Any);
708             var last = sequence.Last(x => x != Constants.Any);
709             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.EnsureEachLinkExists(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);

```

```

759         matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
760             ↪ x)); // OrderBy is a Hack
761         return filteredResults;
762     }
763     return new HashSet<ulong>();
764 }
765
766 // Does not work
767 //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
768     ↪ params ulong[] sequence)
769 //{
770 //    var visited = new HashSet<ulong>();
771 //    var results = new HashSet<ulong>();
772 //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
773     ↪ true; }, readAsElements);
774 //    var last = sequence.Length - 1;
775 //    for (var i = 0; i < last; i++)
776 //    {
777 //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
778 //    }
779 //    return results;
780 //}
781
782 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
783 {
784     return _sync.ExecuteReadOperation(() =>
785     {
786         if (sequence.Length > 0)
787         {
788             Links.EnsureEachLinkExists(sequence);
789             //var firstElement = sequence[0];
790             //if (sequence.Length == 1)
791             //{
792             //    //results.Add(firstElement);
793             //    return results;
794             //}
795             //if (sequence.Length == 2)
796             //{
797             //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
798             //    //if (doublet != Doublets.Links.Null)
799             //    //    results.Add(doublet);
800             //    return results;
801             //}
802             //var lastElement = sequence[sequence.Length - 1];
803             //Func<ulong, bool> handler = x =>
804             //{
805             //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
806             //        ↪ results.Add(x);
807             //    return true;
808             //};
809             //if (sequence.Length >= 2)
810             //    StepRight(handler, sequence[0], sequence[1]);
811             //var last = sequence.Length - 2;
812             //for (var i = 1; i < last; i++)
813             //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
814             //if (sequence.Length >= 3)
815             //    StepLeft(handler, sequence[sequence.Length - 2],
816             //        ↪ sequence[sequence.Length - 1]);
817             //if (sequence.Length == 1)
818             //    throw new NotImplementedException(); // all sequences, containing
819             //    ↪ this element?
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]);

```

```

830         if (results.Count > 0)
831             matches.Add(results);
832         else
833             return results;
834         if (matches.Count == 2)
835         {
836             var merged = new List<ulong>();
837             for (var j = 0; j < matches[0].Count; j++)
838                 for (var k = 0; k < matches[1].Count; k++)
839                     CloseInnerConnections(merged.Add, matches[0][j],
840                     ↪ matches[1][k]);
841             if (merged.Count > 0)
842                 matches = new List<List<ulong>> { merged };
843             else
844                 return new List<ulong>();
845         }
846         if (matches.Count > 0)
847         {
848             var usages = new HashSet<ulong>();
849             for (int i = 0; i < sequence.Length; i++)
850             {
851                 AllUsagesCore(sequence[i], usages);
852             }
853             //for (int i = 0; i < matches[0].Count; i++)
854             //    AllUsagesCore(matches[0][i], usages);
855             //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     return new List<ulong>();
874 });
875 }
876
877 /// <remarks>
878 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
879 /// </remarks>
880 public HashSet<ulong> AllUsages(ulong link)
881 {
882     return _sync.ExecuteReadOperation(() =>
883     {
884         var usages = new HashSet<ulong>();
885         AllUsagesCore(link, usages);
886         return usages;
887     });
888 }
889
890 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
891 ↪ той связи с которой начинался поиск (STTTSSSTT),
892 // причём достаточно одного бита для хранения перехода влево или вправо
893 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
894 {
895     bool handler(ulong doublet)
896     {
897         if (usages.Add(doublet))
898         {
899             AllUsagesCore(doublet, usages);
900         }
901         return true;
902     }
903     Links.Unsync.Each(link, Constants.Any, handler);
904     Links.Unsync.Each(Constants.Any, link, handler);
905 }

```

```

902 public HashSet<ulong> AllBottomUsages(ulong link)
903 {
904     return _sync.ExecuteReadOperation(() =>
905     {
906         var visits = new HashSet<ulong>();
907         var usages = new HashSet<ulong>();
908         AllBottomUsagesCore(link, visits, usages);
909         return usages;
910     });
911 }
912
913 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
914     ↳ usages)
915 {
916     bool handler(ulong doublet)
917     {
918         if (visits.Add(doublet))
919         {
920             AllBottomUsagesCore(doublet, visits, usages);
921         }
922         return true;
923     }
924     if (Links.Unsync.Count(Constants.Any, link) == 0)
925     {
926         usages.Add(link);
927     }
928     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 }

```

```

976 public void CalculateAllUsages2(ulong[] totals)
977 {
978     var calculator = new AllUsagesCalculator2(Links, totals);
979     calculator.Calculate();
980 }
981
982 private class AllUsagesCalculator
983 {
984     private readonly SynchronizedLinks<ulong> _links;
985     private readonly ulong[] _totals;
986
987     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
988     {
989         _links = links;
990         _totals = totals;
991     }
992
993     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
994         ↪ CalculateCore);
995
996     private bool CalculateCore(ulong link)
997     {
998         if (_totals[link] == 0)
999         {
1000             var total = 1UL;
1001             _totals[link] = total;
1002             var visitedChildren = new HashSet<ulong>();
1003             bool linkCalculator(ulong child)
1004             {
1005                 if (link != child && visitedChildren.Add(child))
1006                 {
1007                     total += _totals[child] == 0 ? 1 : _totals[child];
1008                 }
1009                 return true;
1010             }
1011             _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1012             _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1013             _totals[link] = total;
1014         }
1015         return true;
1016     }
1017 }
1018
1019 private class AllUsagesCalculator2
1020 {
1021     private readonly SynchronizedLinks<ulong> _links;
1022     private readonly ulong[] _totals;
1023
1024     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1025     {
1026         _links = links;
1027         _totals = totals;
1028     }
1029
1030     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1031         ↪ CalculateCore);
1032
1033     private bool IsElement(ulong link)
1034     {
1035         // _linksInSequence.Contains(link) ||
1036         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1037             ↪ link;
1038     }
1039
1040     private bool CalculateCore(ulong link)
1041     {
1042         // TODO: Проработать защиту от заикливания
1043         // Основано на SequenceWalker.WalkLeft
1044         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1045         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1046         Func<ulong, bool> isElement = IsElement;
1047         void visitLeaf(ulong parent)
1048         {
1049             if (link != parent)
1050             {
1051                 _totals[parent]++;
1052             }
1053         }
1054         void visitNode(ulong parent)

```

```

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                 // Überprüfung 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>
1177     ↪ intersectWith, HashSet<ulong> usages)
1178     {
1179         _links = links;
1180         _intersectWith = intersectWith;
1181         _usages = usages;
1182         _enter = new HashSet<ulong>(); // защита от зацикливания
1183     }
1184
1185     public bool Collect(ulong link)
1186     {
1187         if (_enter.Add(link))
1188         {
1189             if (_intersectWith.Contains(link))
1190             {
1191                 _usages.Add(link);
1192             }
1193             _links.Unsync.Each(link, _links.Constants.Any, Collect);
1194             _links.Unsync.Each(_links.Constants.Any, link, Collect);
1195         }
1196         return true;
1197     }
1198 }
1199
1200 private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
1201 ↪ right)
1202 {
1203     TryStepLeftUp(handler, left, right);
1204     TryStepRightUp(handler, right, left);
1205 }
1206
1207 private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
1208 ↪ right)
1209 {
1210     // Direct

```



```

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

```

```

1275     {
1276         patternSequence = Simplify(patternSequence);
1277         if (patternSequence.Length > 0)
1278         {
1279             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1280             var uniqueSequenceElements = new HashSet<ulong>();
1281             for (var i = 0; i < patternSequence.Length; i++)
1282             {
1283                 if (patternSequence[i] != Constants.Any && patternSequence[i] !=
1284                     ↪ ZeroOrMany)
1285                 {
1286                     uniqueSequenceElements.Add(patternSequence[i]);
1287                 }
1288             }
1289             var results = new HashSet<ulong>();
1290             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1291             {
1292                 AllUsagesCore(uniqueSequenceElement, results);
1293             }
1294             var filteredResults = new HashSet<ulong>();
1295             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1296             matcher.AddAllPatternMatchedToResults(results);
1297             return filteredResults;
1298         }
1299         return new HashSet<ulong>();
1300     });
1301 }
1302
1303 // Найти все возможные связи между указанным списком связей.
1304 // Находит связи между всеми указанными связями в любом порядке.
1305 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
1306 ↪ несколько раз в последовательности)
1307 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1308 {
1309     return _sync.ExecuteReadOperation(() =>
1310     {
1311         var results = new HashSet<ulong>();
1312         if (linksToConnect.Length > 0)
1313         {
1314             Links.EnsureEachLinkExists(linksToConnect);
1315             AllUsagesCore(linksToConnect[0], results);
1316             for (var i = 1; i < linksToConnect.Length; i++)
1317             {
1318                 var next = new HashSet<ulong>();
1319                 AllUsagesCore(linksToConnect[i], next);
1320                 results.IntersectWith(next);
1321             }
1322             return results;
1323         }
1324     });
1325 }
1326
1327 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1328 {
1329     return _sync.ExecuteReadOperation(() =>
1330     {
1331         var results = new HashSet<ulong>();
1332         if (linksToConnect.Length > 0)
1333         {
1334             Links.EnsureEachLinkExists(linksToConnect);
1335             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1336             collector1.Collect(linksToConnect[0]);
1337             var next = new HashSet<ulong>();
1338             for (var i = 1; i < linksToConnect.Length; i++)
1339             {
1340                 var collector = new AllUsagesCollector(Links.Unsync, next);
1341                 collector.Collect(linksToConnect[i]);
1342                 results.IntersectWith(next);
1343                 next.Clear();
1344             }
1345             return results;
1346         }
1347     });
1348 }
1349
1350 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1351 {
1352     return _sync.ExecuteReadOperation(() =>

```

```

1351 {
1352     var results = new HashSet<ulong>();
1353     if (linksToConnect.Length > 0)
1354     {
1355         Links.EnsureEachLinkExists(linksToConnect);
1356         var collector1 = new AllUsagesCollector(Links, results);
1357         collector1.Collect(linksToConnect[0]);
1358         //AllUsagesCore(linksToConnect[0], results);
1359         for (var i = 1; i < linksToConnect.Length; i++)
1360         {
1361             var next = new HashSet<ulong>();
1362             var collector = new AllUsagesIntersectingCollector(Links, results, next);
1363             collector.Collect(linksToConnect[i]);
1364             //AllUsagesCore(linksToConnect[i], next);
1365             //results.IntersectWith(next);
1366             results = next;
1367         }
1368     }
1369     return results;
1370 });
1371 }
1372
1373 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1374 {
1375     return _sync.ExecuteReadOperation(() =>
1376     {
1377         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1378         ↪ BitArray((int)_links.Total + 1);
1379         if (linksToConnect.Length > 0)
1380         {
1381             Links.EnsureEachLinkExists(linksToConnect);
1382             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1383             collector1.Collect(linksToConnect[0]);
1384             for (var i = 1; i < linksToConnect.Length; i++)
1385             {
1386                 var next = new BitString((long)Links.Unsync.Count() + 1); //new
1387                 ↪ BitArray((int)_links.Total + 1);
1388                 var collector = new AllUsagesCollector2(Links.Unsync, next);
1389                 collector.Collect(linksToConnect[i]);
1390                 results = results.And(next);
1391             }
1392         }
1393         return results.GetSetUInt64Indices();
1394     });
1395 }
1396
1397 private static ulong[] Simplify(ulong[] sequence)
1398 {
1399     // Считаем новый размер последовательности
1400     long newLength = 0;
1401     var zeroOrManyStepped = false;
1402     for (var i = 0; i < sequence.Length; i++)
1403     {
1404         if (sequence[i] == ZeroOrMany)
1405         {
1406             if (zeroOrManyStepped)
1407             {
1408                 continue;
1409             }
1410             zeroOrManyStepped = true;
1411         }
1412         else
1413         {
1414             //if (zeroOrManyStepped) Is it efficient?
1415             zeroOrManyStepped = false;
1416         }
1417         newLength++;
1418     }
1419     // Строим новую последовательность
1420     zeroOrManyStepped = false;
1421     var newSequence = new ulong[newLength];
1422     long j = 0;
1423     for (var i = 0; i < sequence.Length; i++)
1424     {
1425         //var current = zeroOrManyStepped;
1426         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1427         //if (current && zeroOrManyStepped)
1428         //    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>();
1457
1458 public void Prediction()
1459 {
1460     //_links
1461     //_sequences
1462 }
1463
1464 #region From Triplets
1465
1466 //public static void DeleteSequence(Link sequence)
1467 //{
1468 //}
1469
1470 public List<ulong> CollectMatchingSequences(ulong[] links)
1471 {
1472     if (links.Length == 1)
1473     {
1474         throw new Exception("Подпоследовательности с одним элементом не
1475         ↪ поддерживаются.");
1476     }
1477     var leftBound = 0;
1478     var rightBound = links.Length - 1;
1479     var left = links[leftBound++];
1480     var right = links[rightBound--];
1481     var results = new List<ulong>();
1482     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1483     return results;
1484 }
1485
1486 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1487     ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1488 {
1489     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1490     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1491     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1492     {
1493         var nextLeftLink = middleLinks[leftBound];
1494         var elements = GetRightElements(leftLink, nextLeftLink);
1495         if (leftBound <= rightBound)
1496         {
1497             for (var i = elements.Length - 1; i >= 0; i--)
1498             {
1499                 var element = elements[i];
1500                 if (element != 0)
1501                 {
1502                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
1503                     ↪ rightLink, rightBound, ref results);
1504                 }
1505             }
1506         }
1507     }
1508 }

```

```

1502     }
1503     else
1504     {
1505         for (var i = elements.Length - 1; i >= 0; i--)
1506         {
1507             var element = elements[i];
1508             if (element != 0)
1509             {
1510                 results.Add(element);
1511             }
1512         }
1513     }
1514 }
1515 else
1516 {
1517     var nextRightLink = middleLinks[rightBound];
1518     var elements = GetLeftElements(rightLink, nextRightLink);
1519     if (leftBound <= rightBound)
1520     {
1521         for (var i = elements.Length - 1; i >= 0; i--)
1522         {
1523             var element = elements[i];
1524             if (element != 0)
1525             {
1526                 CollectMatchingSequences(leftLink, leftBound, middleLinks,
1527                                         ↪ elements[i], rightBound - 1, ref results);
1528             }
1529         }
1530     }
1531     else
1532     {
1533         for (var i = elements.Length - 1; i >= 0; i--)
1534         {
1535             var element = elements[i];
1536             if (element != 0)
1537             {
1538                 results.Add(element);
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
    ↪ == Net.And &&
1583 {
1584     result[offset + 1] = couple;
1585     if (++added == 2)
1586     {
1587         return false;
1588     }
1589 }
1590 }
1591 return true;
1592 });
1593 return added > 0;
1594 }
1595
1596 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1597 {
1598     var result = new ulong[5];
1599     TryStepLeft(startLink, leftLink, result, 0);
1600     Links.Each(startLink, Constants.Any, couple =>
1601     {
1602         if (couple != startLink)
1603         {
1604             if (TryStepLeft(couple, leftLink, result, 2))
1605             {
1606                 return false;
1607             }
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         }
1644         return true;
1645     });
1646     return added > 0;
1647 }
1648
1649 #endregion
1650
1651 #region Walkers
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;
1687     _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1688     ↳ _sequences.Constants.Any && x != ZeroOrMany));
1689     _results = results;
1690     _pattern = CreateDetailedPattern();
1691 }
1692
1693 protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
1694     ↳ base.IsElement(link);
1695
1696 public bool PatternMatch(LinkIndex sequenceToMatch)
1697 {
1698     _patternPosition = 0;
1699     _sequencePosition = 0;
1700     foreach (var part in Walk(sequenceToMatch))
1701     {
1702         if (!PatternMatchCore(part))
1703         {
1704             break;
1705         }
1706     }
1707     return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1708     ↳ - 1 && _pattern[_patternPosition].Start == 0);
1709 }
1710
1711 private List<PatternBlock> CreateDetailedPattern()
1712 {
1713     var pattern = new List<PatternBlock>();
1714     var patternBlock = new PatternBlock();
1715     for (var i = 0; i < _patternSequence.Length; i++)
1716     {
1717         if (patternBlock.Type == PatternBlockType.Undefined)
1718         {
1719             if (_patternSequence[i] == _sequences.Constants.Any)
1720             {
1721                 patternBlock.Type = PatternBlockType.Gap;
1722                 patternBlock.Start = 1;
1723                 patternBlock.Stop = 1;
1724             }
1725             else if (_patternSequence[i] == ZeroOrMany)
1726             {
1727                 patternBlock.Type = PatternBlockType.Gap;
1728                 patternBlock.Start = 0;
1729                 patternBlock.Stop = long.MaxValue;
1730             }
1731             else
1732             {
1733                 patternBlock.Type = PatternBlockType.Elements;
1734                 patternBlock.Start = i;
1735                 patternBlock.Stop = i;
1736             }
1737         }
1738     }

```

```

1733     }
1734     else if (patternBlock.Type == PatternBlockType.Elements)
1735     {
1736         if (_patternSequence[i] == _sequences.Constants.Any)
1737         {
1738             pattern.Add(patternBlock);
1739             patternBlock = new PatternBlock
1740             {
1741                 Type = PatternBlockType.Gap,
1742                 Start = 1,
1743                 Stop = 1
1744             };
1745         }
1746         else if (_patternSequence[i] == ZeroOrMany)
1747         {
1748             pattern.Add(patternBlock);
1749             patternBlock = new PatternBlock
1750             {
1751                 Type = PatternBlockType.Gap,
1752                 Start = 0,
1753                 Stop = long.MaxValue
1754             };
1755         }
1756         else
1757         {
1758             patternBlock.Stop = i;
1759         }
1760     }
1761     else // patternBlock.Type == PatternBlockType.Gap
1762     {
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 }
1787 if (patternBlock.Type != PatternBlockType.Undefined)
1788 {
1789     pattern.Add(patternBlock);
1790 }
1791 return pattern;
1792 }
1793
1794 // match: search for regexp anywhere in text
1795 //int match(char* regexp, char* text)
1796 //{
1797 //    do
1798 //    {
1799 //        } while (*text++ != '\0');
1800 //    return 0;
1801 //}
1802
1803 // matchhere: search for regexp at beginning of text
1804 //int matchhere(char* regexp, char* text)
1805 //{
1806 //    if (regexp[0] == '\0')
1807 //        return 1;
1808 //    if (regexp[1] == '*')
1809 //        return matchstar(regexp[0], regexp + 2, text);
1810 //    if (regexp[0] == '$' && regexp[1] == '\0')
1811 //        return *text == '\0';

```



```

1812 // if (*text != '\0' && (regex[0] == '.' || regex[0] == *text))
1813 //     return matchhere(regex + 1, text + 1);
1814 // return 0;
1815 //}
1816
1817 // matchstar: search for c*regex at beginning of text
1818 //int matchstar(int c, char* regex, char* text)
1819 //{
1820 //    do
1821 //    { /* a * matches zero or more instances */
1822 //        if (matchhere(regex, 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;
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 }

```

```

1889     }
1890 }
1891 }
1892 else // currentPatternBlock.Type == PatternBlockType.Elements
1893 {
1894     var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1895     if (_patternSequence[patternElementPosition] != element)
1896     {
1897         return false; // Соответствие невозможно
1898     }
1899     if (patternElementPosition == currentPatternBlock.Stop)
1900     {
1901         _patternPosition++;
1902         _sequencePosition = 0;
1903     }
1904     else
1905     {
1906         _sequencePosition++;
1907     }
1908 }
1909 return true;
1910 //if (_patternSequence[_patternPosition] != element)
1911 //    return false;
1912 //else
1913 //{
1914 //    _sequencePosition++;
1915 //    _patternPosition++;
1916 //    return true;
1917 //}
1918 ///////
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             }
50         }
51     }
52 }

```

```

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
57                 ↪ link.");
58             }
59         }
60         if (MarkedSequenceMatcher == null)
61         {
62             MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
63             ↪ SequenceMarkerLink);
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
74                     ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
75                     ↪ MarkedSequenceMatcher);
76                 }
77                 else
78                 {
79                     totalSequenceSymbolFrequencyCounter = new
80                     ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
81                 }
82                 var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
83                 ↪ totalSequenceSymbolFrequencyCounter);
84                 var compressingConverter = new CompressingConverter<TLink>(links,
85                 ↪ balancedVariantConverter, doubletFrequenciesCache);
86                 LinksToSequenceConverter = compressingConverter;
87             }
88         }
89         else
90         {
91             if (LinksToSequenceConverter == null)
92             {
93                 LinksToSequenceConverter = balancedVariantConverter;
94             }
95         }
96         if (UseIndex && Index == null)
97         {
98             Index = new SequenceIndex<TLink>(links);
99         }
100         if (Walker == null)
101         {
102             Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
103         }
104     }
105     public void ValidateOptions()
106     {
107         if (UseGarbageCollection && !UseSequenceMarker)
108         {
109             throw new NotSupportedException("To use garbage collection UseSequenceMarker
110             ↪ option must be on.");
111         }
112     }
113 }

```

./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 }

```

```

18     [MethodImpl(MethodImplOptions.AggressiveInlining)]
19     protected override TLink GetNextElementAfterPush(TLink element) =>
20         ↪ Links.GetTarget(element);
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected override IEnumerable<TLink> WalkContents(TLink element)
24     {
25         var parts = Links.GetLink(element);
26         var start = Links.Constants.IndexPart + 1;
27         for (var i = parts.Count - 1; i >= start; i--)
28         {
29             var part = parts[i];
30             if (IsElement(part))
31             {
32                 yield return part;
33             }
34         }
35     }
36 }
37 }

```

./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                     {

```

```

55         nextArray[doubletOffset] = candidate;
56     }
57     else
58     {
59         var link = Links.GetLink(candidate);
60         var linkSource = Links.GetSource(link);
61         var linkTarget = Links.GetTarget(link);
62         nextArray[doubletOffset] = linkSource;
63         nextArray[doubletOffset + 1] = linkTarget;
64         if (!hasElements)
65         {
66             hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
67         }
68     }
69 }
70 #if USEARRAYPOOL
71     if (array.Length > 1)
72     {
73         ArrayPool.Free(array);
74     }
75 #endif
76     array = nextArray;
77 }
78 while (hasElements);
79 var filledElementsCount = CountFilledElements(array);
80 if (filledElementsCount == array.Length)
81 {
82     return array;
83 }
84 else
85 {
86     return CopyFilledElements(array, filledElementsCount);
87 }
88 }
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
92 {
93     var finalArray = new TLink[filledElementsCount];
94     for (int i = 0, j = 0; i < array.Length; i++)
95     {
96         if (!_equalityComparer.Equals(array[i], default))
97         {
98             finalArray[j] = array[i];
99             j++;
100         }
101     }
102     #if USEARRAYPOOL
103         ArrayPool.Free(array);
104     #endif
105     return finalArray;
106 }
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 private static int CountFilledElements(TLink[] array)
110 {
111     var count = 0;
112     for (var i = 0; i < array.Length; i++)
113     {
114         if (!_equalityComparer.Equals(array[i], default))
115         {
116             count++;
117         }
118     }
119     return count;
120 }
121 }
122 }

```

./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>
        ↳ isElement) : base(links, stack, isElement) { }
13
14     public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
        ↳ stack, links.IsPartialPoint) { }
15
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     protected override TLink GetNextElementAfterPop(TLink element) =>
        ↳ Links.GetTarget(element);
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     protected override TLink GetNextElementAfterPush(TLink element) =>
        ↳ Links.GetSource(element);
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected override IEnumerable<TLink> WalkContents(TLink element)
24     {
25         var parts = Links.GetLink(element);
26         for (var i = Links.Constants.IndexPart + 1; i < parts.Count; i++)
27         {
28             var part = parts[i];
29             if (IsElement(part))
30             {
31                 yield return part;
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>,
        ↳ ISequenceWalker<TLink>
11     {
12         private readonly IStack<TLink> _stack;
13         private readonly Func<TLink, bool> _isElement;
14
15         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↳ isElement) : base(links)
16         {
17             _stack = stack;
18             _isElement = isElement;
19         }
20
21         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
        ↳ stack, links.IsPartialPoint)
22         {
23         }
24
25         public IEnumerable<TLink> Walk(TLink sequence)
26         {
27             _stack.Clear();
28             var element = sequence;
29             if (IsElement(element))
30             {
31                 yield return element;
32             }
33             else
34             {
35                 while (true)
36                 {
37                     if (IsElement(element))
38                     {
39                         if (_stack.IsEmpty)
40                         {
41                             break;
42                         }
43                         element = _stack.Pop();
44                         foreach (var output in WalkContents(element))

```



```

45         {
46             yield return output;
47         }
48         element = GetNextElementAfterPop(element);
49     }
50     else
51     {
52         _stack.Push(element);
53         element = GetNextElementAfterPush(element);
54     }
55 }
56 }
57 }
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected abstract TLink GetNextElementAfterPop(TLink element);
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected abstract TLink GetNextElementAfterPush(TLink element);
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 protected abstract IEnumerable<TLink> WalkContents(TLink element);
70 }
71 }

```

./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
23             ↳ ReaderWriterLockSynchronization(), links) { }
24
25         public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
26         {
27             SyncRoot = synchronization;
28             Sync = this;
29             Unsync = links;
30             Constants = links.Constants;
31         }
32
33         public TLinkAddress Count(IList<TLinkAddress> restriction) =>
34             ↳ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
35         public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
36             ↳ IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
37             ↳ restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
38         public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
39             ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
40         public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
41             ↳ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
42             ↳ Unsync.Update);
43         public void Delete(IList<TLinkAddress> restrictions) =>
44             ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
45
46         //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
47         ↳ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
48         //{
49         //    if (restriction != null && substitution != null &&
50         ↳ !substitution.EqualTo(restriction))
51         //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
52         ↳ substitution, substitutedHandler, Unsync.Trigger);
53         //
54         //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
55         ↳ substitutedHandler, Unsync.Trigger);
56         //}
57     }
58 }

```

./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 =
15             ↳ Default<LinksConstants<ulong>>.Instance;
16
17         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
18
19         public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong> sequence)
20         {
21             if (sequence == null)
22             {
23                 return;
24             }
25             for (var i = 0; i < sequence.Count; i++)
26             {
27                 if (!links.Exists(sequence[i]))
28                 {
29                     throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
30                         ↳ $"sequence[{i}]");
31                 }
32             }
33
34         public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links, IList<ulong>
35             ↳ sequence)
36         {
37             if (sequence == null)
38             {
39                 return;
40             }
41             for (var i = 0; i < sequence.Count; i++)
42             {
43                 if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
44                 {
45                     throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
46                         ↳ $"sequence[{i}]");
47                 }
48             }
49
50         public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
51         {
52             if (sequence == null)
53             {
54                 return false;
55             }
56             var constants = links.Constants;
57             for (var i = 0; i < sequence.Length; i++)
58             {
59                 if (sequence[i] == constants.Any)
60                 {
61                     return true;
62                 }
63             }
64             return false;
65
66         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
67             ↳ Func<Link<ulong>, bool> isElement, bool renderIndex = false, bool renderDebug =
68             ↳ false)
69         {
70             var sb = new StringBuilder();
71             var visited = new HashSet<ulong>();
72             links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
73                 ↳ innerSb.Append(link.Index), renderIndex, renderDebug);
74             return sb.ToString();
75
76         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
77             ↳ Func<Link<ulong>, bool> isElement, Action<StringBuilder, Link<ulong>> appendElement,
78             ↳ bool renderIndex = false, bool renderDebug = false)
79         {
80

```

```

75     var sb = new StringBuilder();
76     var visited = new HashSet<ulong>();
77     links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
    ↪     renderDebug);
78     return sb.ToString();
79 }
80
81 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)
82 {
83     if (sb == null)
84     {
85         throw new ArgumentNullException(nameof(sb));
86     }
87     if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
    ↪     Constants.Itself)
88     {
89         return;
90     }
91     if (links.Exists(linkIndex))
92     {
93         if (visited.Add(linkIndex))
94         {
95             sb.Append('(');
96             var link = new Link<ulong>(links.GetLink(linkIndex));
97             if (renderIndex)
98             {
99                 sb.Append(link.Index);
100                 sb.Append(':');
101             }
102             if (link.Source == link.Index)
103             {
104                 sb.Append(link.Index);
105             }
106             else
107             {
108                 var source = new Link<ulong>(links.GetLink(link.Source));
109                 if (isElement(source))
110                 {
111                     appendElement(sb, source);
112                 }
113                 else
114                 {
115                     links.AppendStructure(sb, visited, source.Index, isElement,
    ↪                     appendElement, renderIndex);
116                 }
117             }
118             sb.Append(' ');
119             if (link.Target == link.Index)
120             {
121                 sb.Append(link.Index);
122             }
123             else
124             {
125                 var target = new Link<ulong>(links.GetLink(link.Target));
126                 if (isElement(target))
127                 {
128                     appendElement(sb, target);
129                 }
130                 else
131                 {
132                     links.AppendStructure(sb, visited, target.Index, isElement,
    ↪                     appendElement, renderIndex);
133                 }
134             }
135             sb.Append(')');
136         }
137         else
138         {
139             if (renderDebug)
140             {
141                 sb.Append('*');
142             }
143             sb.Append(linkIndex);
144         }
145     }

```

```

146         else
147         {
148             if (renderDebug)
149             {
150                 sb.Append('~');
151             }
152             sb.Append(linkIndex);
153         }
154     }
155 }
156 }

```

./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
124     ///    со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
125     ///    ↪ пересечениями идентификаторов.
126     ///
127     /// Где хранить промежуточный список транзакций?
128     ///
129     /// В оперативной памяти:
130     /// Минусы:
131     /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
132     ///    так как нужно отдельно выделять память под список трансформаций.
133     /// 2. Выделенной оперативной памяти может не хватить, в том случае,
134     ///    если транзакция использует слишком много трансформаций.
135     ///    -> Можно использовать жёсткий диск для слишком длинных транзакций.
136     ///    -> Максимальный размер списка трансформаций можно ограничить / задать
137     ///    ↪ константой.
138     /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
139     ///    ↪ создавая задержку.
140     ///
141     /// На жёстком диске:
142     /// Минусы:
143     /// 1. Длительный отклик, на запись каждой трансформации.
144     /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.

```

```

137     /// -> Это может решаться упаковкой/исключением дублирующих операций.
138     /// -> Также это может решаться тем, что короткие транзакции вообще
139     /// не будут записываться в случае отката.
140     /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
    ↪ операции (трансформации)
141     /// будут записаны в лог.
142     ///
143     </remarks>
144     public class Transaction : DisposableBase
145     {
146         private readonly Queue<Transition> _transitions;
147         private readonly UInt64LinksTransactionsLayer _layer;
148         public bool IsCommitted { get; private set; }
149         public bool IsReverted { get; private set; }
150
151         public Transaction(UInt64LinksTransactionsLayer layer)
152         {
153             _layer = layer;
154             if (_layer._currentTransactionId != 0)
155             {
156                 throw new NotSupportedException("Nested transactions not supported.");
157             }
158             IsCommitted = false;
159             IsReverted = false;
160             _transitions = new Queue<Transition>();
161             SetCurrentTransaction(layer, this);
162         }
163
164         public void Commit()
165         {
166             EnsureTransactionAllowsWriteOperations(this);
167             while (_transitions.Count > 0)
168             {
169                 var transition = _transitions.Dequeue();
170                 _layer._transitions.Enqueue(transition);
171             }
172             _layer._lastCommittedTransactionId = _layer._currentTransactionId;
173             IsCommitted = true;
174         }
175
176         private void Revert()
177         {
178             EnsureTransactionAllowsWriteOperations(this);
179             var transitionsToRevert = new Transition[_transitions.Count];
180             _transitions.CopyTo(transitionsToRevert, 0);
181             for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
182             {
183                 _layer.RevertTransition(transitionsToRevert[i]);
184             }
185             IsReverted = true;
186         }
187
188         public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
    ↪ Transaction transaction)
189         {
190             layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
191             layer._currentTransactionTransitions = transaction._transitions;
192             layer._currentTransaction = transaction;
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 закоммиченную 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 {
272     var createdLinkIndex = Links.Create();
273     var createdLink = new Link<ulong>(Links.GetLink(createdLinkIndex));
274     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
275         ↳ default, createdLink));
276     return createdLinkIndex;
277 }
278
279 public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
280 {
281     var linkIndex = restrictions[Constants.IndexPart];
282     var beforeLink = new Link<ulong>(Links.GetLink(linkIndex));
283     linkIndex = Links.Update(restrictions, substitution);
284     var afterLink = new Link<ulong>(Links.GetLink(linkIndex));
285     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
286         ↳ beforeLink, afterLink));
287     return linkIndex;
288 }
289
290 public override void Delete(IList<ulong> restrictions)
291 {
292     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         }
23     }
24 }

```

```

19     _index = index;
20     _listToSequenceLinkConverter = listToSequenceLinkConverter;
21     _unicodeSequenceMarker = unicodeSequenceMarker;
22 }
23
24 public TLink Convert(string source)
25 {
26     var elements = new TLink[source.Length];
27     for (int i = 0; i < source.Length; i++)
28     {
29         elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
30     }
31     _index.Add(elements);
32     var sequence = _listToSequenceLinkConverter.Convert(elements);
33     return Links.GetOrCreate(sequence, _unicodeSequenceMarker);
34 }
35 }
36 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Globalization;
4  using System.Runtime.CompilerServices;
5  using System.Text;
6  using Platform.Data.Sequences;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Unicode
11 {
12     public class UnicodeMap
13     {
14         public static readonly ulong FirstCharLink = 1;
15         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
16         public static readonly ulong MapSize = 1 + char.MaxValue;
17
18         private readonly ILinks<ulong> _links;
19         private bool _initialized;
20
21         public UnicodeMap(ILinks<ulong> links) => _links = links;
22
23         public static UnicodeMap InitNew(ILinks<ulong> links)
24         {
25             var map = new UnicodeMap(links);
26             map.Init();
27             return map;
28         }
29
30         public void Init()
31         {
32             if (_initialized)
33             {
34                 return;
35             }
36             _initialized = true;
37             var firstLink = _links.CreatePoint();
38             if (firstLink != FirstCharLink)
39             {
40                 _links.Delete(firstLink);
41             }
42             else
43             {
44                 for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
45                 {
46                     // From NIL to It (NIL -> Character) transformation meaning, (or infinite
47                     //   ↳ amount of NIL characters before actual Character)
48                     var createdLink = _links.CreatePoint();
49                     _links.Update(createdLink, firstLink, createdLink);
50                     if (createdLink != i)
51                     {
52                         throw new InvalidOperationException("Unable to initialize UTF 16
53                         ↳ table.");
54                     }
55                 }
56             }
57         }
58     }
59 }

```

// 0- null link
// 1- nil character (0 character)

```

59 // ...
60 // 65536 (0(1) + 65535 = 65536 possible values)
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 public static ulong FromCharToLink(char character) => (ulong)character + 1;
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 public static char FromLinkToChar(ulong link) => (char)(link - 1);
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 public static bool IsCharLink(ulong link) => link <= MapSize;
70
71 public static string FromLinksToString(IList<ulong> linksList)
72 {
73     var sb = new StringBuilder();
74     for (int i = 0; i < linksList.Count; i++)
75     {
76         sb.Append(FromLinkToChar(linksList[i]));
77     }
78     return sb.ToString();
79 }
80
81 public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
82 {
83     var sb = new StringBuilder();
84     if (links.Exists(link))
85     {
86         StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
87             x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
88             ↪ element =>
89             {
90                 sb.Append(FromLinkToChar(element));
91                 return true;
92             }
93     }
94     return sb.ToString();
95 }
96
97 public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
98     ↪ chars.Length);
99
100 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
101 {
102     // char array to ulong array
103     var linksSequence = new ulong[count];
104     for (var i = 0; i < count; i++)
105     {
106         linksSequence[i] = FromCharToLink(chars[i]);
107     }
108     return linksSequence;
109 }
110
111 public static ulong[] FromStringToLinkArray(string sequence)
112 {
113     // char array to ulong array
114     var linksSequence = new ulong[sequence.Length];
115     for (var i = 0; i < sequence.Length; i++)
116     {
117         linksSequence[i] = FromCharToLink(sequence[i]);
118     }
119     return linksSequence;
120 }
121
122 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
123 {
124     var result = new List<ulong[]>();
125     var offset = 0;
126     while (offset < sequence.Length)
127     {
128         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
129         var relativeLength = 1;
130         var absoluteLength = offset + relativeLength;
131         while (absoluteLength < sequence.Length &&
132             currentCategory ==
133             ↪ CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
134         {
135             relativeLength++;
136             absoluteLength++;
137         }
138     }
139 }

```

```

135         // char array to ulong array
136         var innerSequence = new ulong[relativeLength];
137         var maxLength = offset + relativeLength;
138         for (var i = offset; i < maxLength; i++)
139         {
140             innerSequence[i - offset] = FromCharToLink(sequence[i]);
141         }
142         result.Add(innerSequence);
143         offset += relativeLength;
144     }
145     return result;
146 }
147
148 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
149 {
150     var result = new List<ulong[]>();
151     var offset = 0;
152     while (offset < array.Length)
153     {
154         var relativeLength = 1;
155         if (array[offset] <= LastCharLink)
156         {
157             var currentCategory =
158                 CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
159             var absoluteLength = offset + relativeLength;
160             while (absoluteLength < array.Length &&
161                 array[absoluteLength] <= LastCharLink &&
162                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
163                     array[absoluteLength])))
164             {
165                 relativeLength++;
166                 absoluteLength++;
167             }
168         }
169         else
170         {
171             var absoluteLength = offset + relativeLength;
172             while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
173             {
174                 relativeLength++;
175                 absoluteLength++;
176             }
177         }
178         // copy array
179         var innerSequence = new ulong[relativeLength];
180         var maxLength = offset + relativeLength;
181         for (var i = offset; i < maxLength; i++)
182         {
183             innerSequence[i - offset] = array[i];
184         }
185         result.Add(innerSequence);
186         offset += relativeLength;
187     }
188     return result;
189 }

```

./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 }
```

./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)
```

```

15     {
16         _numberToAddressConverter = numberToAddressConverter;
17         _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
18     }
19
20     public char Convert(TLink source)
21     {
22         if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
23         {
24             throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
25                 ↪ not a unicode symbol.");
26         }
27         return (char)(ushort)(Integer<TLink>)_numberToAddressConverter.Convert(Links.GetSource(source));
28     }
29 }

```

./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         private static bool Equals2<T>(T x, T y) => x.Equals(y);
19         private static bool Equals3(ulong x, ulong y) => x == y;
20
21         [Fact]
22         public static void EqualsPerformanceTest()
23         {
24             const int N = 1000000;
25
26             ulong x = 10;
27             ulong y = 500;
28
29             bool result = false;
30
31             var ts1 = Performance.Measure(() =>
32             {
33                 for (int i = 0; i < N; i++)
34                 {
35                     result = Equals1(x, y);
36                 }
37             });
38
39             var ts2 = Performance.Measure(() =>
40             {
41                 for (int i = 0; i < N; i++)
42                 {
43                     result = Equals2(x, y);
44                 }
45             });
46
47             var ts3 = Performance.Measure(() =>
48             {
49                 for (int i = 0; i < N; i++)
50                 {
51                     result = Equals3(x, y);
52                 }
53             });
54
55             var equalityComparer1 = EqualityComparer<ulong>.Default;
56
57             var ts4 = Performance.Measure(() =>
58             {
59                 for (int i = 0; i < N; i++)
60                 {
61                     result = equalityComparer1.Equals(x, y);
62                 }
63             });
64
65             var equalityComparer2 = new UInt64EqualityComparer();
66
67             var ts5 = Performance.Measure(() =>
68             {
69                 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 }

```

```

39     private static void Using<TLink>(Action<ILinks<TLink>> action)
40     {
41         using (var scope = new Scope<Types<HeapResizableDirectMemory,
42             ↳ ResizableDirectMemoryLinks<TLink>>>())
43         {
44             action(scope.Use<ILinks<TLink>>());
45         }
46     }
47 }

```

./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
20 namespace Platform.Data.Doublets.Tests
21 {
22     public static class OptimalVariantSequenceTests
23     {
24         private const string SequenceExample = "зеленела зелёная зелень";
25
26         [Fact]
27         public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
28         {
29             using (var scope = new TempLinksTestScope(useSequences: false))
30             {
31                 var links = scope.Links;
32                 var constants = links.Constants;
33
34                 links.UseUnicode();
35
36                 var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
37
38                 var meaningRoot = links.CreatePoint();
39                 var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
40                 var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
41                 var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
42                     ↳ constants.Itself);
43
44                 var unaryNumberToAddressConverter = new
45                     ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);

```

```

44     var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
45     var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
    ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
46     var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
    ↪ frequencyPropertyMarker, frequencyMarker);
47     var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
    ↪ frequencyPropertyOperator, frequencyIncrementer);
48     var linkToItsFrequencyNumberConverter = new
    ↪ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
    ↪ unaryNumberToAddressConverter);
49     var sequenceToItsLocalElementLevelsConverter = new
    ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
    ↪ linkToItsFrequencyNumberConverter);
50     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
    ↪ sequenceToItsLocalElementLevelsConverter);
51
52     var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
    ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
53
54     ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
    ↪ index, optimalVariantConverter);
55 }
56 }
57
58 [Fact]
59 public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
60 {
61     using (var scope = new TempLinksTestScope(useSequences: false))
62     {
63         var links = scope.Links;
64
65         links.UseUnicode();
66
67         var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
68
69         var linksToFrequencies = new Dictionary<ulong, ulong>();
70
71         var totalSequenceSymbolFrequencyCounter = new
    ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
72
73         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
    ↪ totalSequenceSymbolFrequencyCounter);
74
75         var index = new
    ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
76         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
    ↪ ncyNumberConverter<ulong>(linkFrequenciesCache);
77
78         var sequenceToItsLocalElementLevelsConverter = new
    ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
    ↪ linkToItsFrequencyNumberConverter);
79         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
    ↪ sequenceToItsLocalElementLevelsConverter);
80
81         var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
    ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
82
83         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
    ↪ index, optimalVariantConverter);
84     }
85 }
86
87 private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
    ↪ SequenceToItsLocalElementLevelsConverter<ulong>
    ↪ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
    ↪ OptimalVariantConverter<ulong> optimalVariantConverter)
88 {
89     index.Add(sequence);
90
91     var optimalVariant = optimalVariantConverter.Convert(sequence);
92
93     var readSequence1 = sequences.ToList(optimalVariant);
94
95     Assert.True(sequence.SequenceEqual(readSequence1));
96 }
97
98 [Fact]
99 public static void SavedSequencesOptimizationTest()

```

```

100 {
101     using (var memory = new HeapResizableDirectMemory())
102     using (var disposableLinks = new UInt64ResizableDirectMemoryLinks(memory))
103     {
104         var links = new UInt64Links(disposableLinks);
105
106         var meaningRoot = links.CreatePoint();
107         var unaryOne = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
108
109         var linksToFrequencies = new Dictionary<ulong, ulong>();
110         var totalSequenceSymbolFrequencyCounter = new
111             ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links);
112         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
113             ↳ totalSequenceSymbolFrequencyCounter);
114         var index = new
115             ↳ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
116         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
117         var sequenceToItsLocalElementLevelsConverter = new
118             ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
119             ↳ linkToItsFrequencyNumberConverter);
120         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
121             ↳ sequenceToItsLocalElementLevelsConverter);
122
123         var sequencesOptions = new SequencesOptions<ulong>()
124         {
125
126         };
127
128         var sequences = new Sequences.Sequences(new SynchronizedLinks<ulong>(links));
129
130     }
131     // create some sequences
132     // get list of sequences links
133     // for each sequence link
134     //     create new sequence version
135     //     if new sequence is not the same as sequence link
136     //         delete sequence link
137     //         collect garbadage
138 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Linq;
5  using Xunit;
6  using Platform.Data.Sequences;
7  using Platform.Data.Doublets.Sequences.Converters;
8  using Platform.Data.Doublets.Sequences.Walkers;
9  using Platform.Data.Doublets.Sequences;
10
11 namespace Platform.Data.Doublets.Tests
12 {
13     public static class ReadSequenceTests
14     {
15         [Fact]
16         public static void ReadSequenceTest()
17         {
18             const long sequenceLength = 2000;
19
20             using (var scope = new TempLinksTestScope(useSequences: false))
21             {
22                 var links = scope.Links;
23                 var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong> {
24                     ↳ Walker = new LeveledSequenceWalker<ulong>(links) });
25
26                 var sequence = new ulong[sequenceLength];
27                 for (var i = 0; i < sequenceLength; i++)
28                 {
29                     sequence[i] = links.Create();
30                 }
31
32                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
33
34                 var sw1 = Stopwatch.StartNew();

```

```

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

```

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

```

1  using System.IO;
2  using Xunit;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using Platform.Data.Doublets.ResizableDirectMemory.Specific;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public static class ResizableDirectMemoryLinksTests
10     {
11         private static readonly LinksConstants<ulong> _constants =
12             ↪ Default<LinksConstants<ulong>>.Instance;
13
14         [Fact]
15         public static void BasicFileMappedMemoryTest()
16         {
17             var tempFilename = Path.GetTempFileName();
18             using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(tempFilename))
19             {
20                 memoryAdapter.TestBasicMemoryOperations();
21             }
22             File.Delete(tempFilename);
23
24             [Fact]
25             public static void BasicHeapMemoryTest()
26             {
27                 using (var memory = new
28                     ↪ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
29                 using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
30                     ↪ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
31                 {
32                     memoryAdapter.TestBasicMemoryOperations();
33                 }
34
35                 private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
36                 {
37                     var link = memoryAdapter.Create();
38                     memoryAdapter.Delete(link);
39                 }
40
41                 [Fact]
42                 public static void NonexistentReferencesHeapMemoryTest()
43                 {
44                     using (var memory = new
45                         ↪ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))

```

```

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

```

./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();
50
51                 Assert.True(results1.Count > results2.Length);
52                 Assert.True(sw1.Elapsed > sw2.Elapsed);
53
54                 for (var i = 0; i < sequenceLength; i++)
55                 {
56                     links.Delete(sequence[i]);
57                 }
58
59                 Assert.True(links.Count() == 0);
60             }
61
62             //[Fact]
63             //public void CUDTest()
64             //{
65             //    var tempFilename = Path.GetTempFileName();
66
67             //    const long sequenceLength = 8;
68
69             //    const ulong itself = LinksConstants.Itself;
70
71             //    using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
72             //        ↪ DefaultLinksSizeStep))
73             //    {
74                 using (var links = new Links(memoryAdapter))
75                 {
76                     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 // }

```

[Fact]

```

100
101 public static void AllVariantsSearchTest()
102 {
103     const long sequenceLength = 8;
104
105     using (var scope = new TempLinksTestScope(useSequences: true))
106     {
107         var links = scope.Links;
108         var sequences = scope.Sequences;
109
110         var sequence = new ulong[sequenceLength];
111         for (var i = 0; i < sequenceLength; i++)
112         {
113             sequence[i] = links.Create();
114         }
115
116         var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
117
118         //for (int i = 0; i < createResults.Length; i++)
119         //    sequences.Create(createResults[i]);
120
121         var sw0 = Stopwatch.StartNew();
122         var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
123
124         var sw1 = Stopwatch.StartNew();
125         var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
126
127         var sw2 = Stopwatch.StartNew();
128         var searchResults2 = sequences.Each1(sequence); sw2.Stop();
129
130         var sw3 = Stopwatch.StartNew();
131         var searchResults3 = sequences.Each(sequence.ConvertToRestrictionsValues());
132         ↪ sw3.Stop();
133
134         var intersection0 = createResults.Intersect(searchResults0).ToList();
135         Assert.True(intersection0.Count == searchResults0.Count);
136         Assert.True(intersection0.Count == createResults.Length);
137
138         var intersection1 = createResults.Intersect(searchResults1).ToList();
139         Assert.True(intersection1.Count == searchResults1.Count);
140         Assert.True(intersection1.Count == createResults.Length);
141
142         var intersection2 = createResults.Intersect(searchResults2).ToList();
143         Assert.True(intersection2.Count == searchResults2.Count);
144         Assert.True(intersection2.Count == createResults.Length);
145
146         var intersection3 = createResults.Intersect(searchResults3).ToList();
147         Assert.True(intersection3.Count == searchResults3.Count);
148         Assert.True(intersection3.Count == createResults.Length);
149
150         for (var i = 0; i < sequenceLength; i++)
151         {
152             links.Delete(sequence[i]);
153         }

```



```

154     }
155 }
156
157 [Fact]
158 public static void BalancedVariantSearchTest()
159 {
160     const long sequenceLength = 200;
161
162     using (var scope = new TempLinksTestScope(useSequences: true))
163     {
164         var links = scope.Links;
165         var sequences = scope.Sequences;
166
167         var sequence = new ulong[sequenceLength];
168         for (var i = 0; i < sequenceLength; i++)
169         {
170             sequence[i] = links.Create();
171         }
172
173         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
174
175         var sw1 = Stopwatch.StartNew();
176         var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
177
178         var sw2 = Stopwatch.StartNew();
179         var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
180
181         var sw3 = Stopwatch.StartNew();
182         var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
183
184         // На количестве в 200 элементов это будет занимать вечность
185         //var sw4 = Stopwatch.StartNew();
186         //var searchResults4 = sequences.Each(sequence); sw4.Stop();
187
188         Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
189
190         Assert.True(searchResults3.Count == 1 && balancedVariant ==
191             ↪ searchResults3.First());
192
193         //Assert.True(sw1.Elapsed < sw2.Elapsed);
194
195         for (var i = 0; i < sequenceLength; i++)
196         {
197             links.Delete(sequence[i]);
198         }
199     }
200 }
201
202 [Fact]
203 public static void AllPartialVariantsSearchTest()
204 {
205     const long sequenceLength = 8;
206
207     using (var scope = new TempLinksTestScope(useSequences: true))
208     {
209         var links = scope.Links;
210         var sequences = scope.Sequences;
211
212         var sequence = new ulong[sequenceLength];
213         for (var i = 0; i < sequenceLength; i++)
214         {
215             sequence[i] = links.Create();
216         }
217
218         var createResults = sequences.CreateAllVariants2(sequence);
219
220         //var createResultsStrings = createResults.Select(x => x + ": " +
221             ↪ sequences.FormatSequence(x)).ToList();
222         //Global.Trash = createResultsStrings;
223
224         var partialSequence = new ulong[sequenceLength - 2];
225
226         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
227
228         var sw1 = Stopwatch.StartNew();
229         var searchResults1 =
230             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
231
232         var sw2 = Stopwatch.StartNew();

```

```

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

```

```

302 {
303     var zeroOrMany = Sequences.Sequences.ZeroOrMany;
304
305     using (var scope = new TempLinksTestScope(useSequences: true))
306     {
307         var links = scope.Links;
308         var sequences = scope.Sequences;
309
310         var e1 = links.Create();
311         var e2 = links.Create();
312
313         var sequence = new[]
314         {
315             e1, e2, e1, e2 // mama / papa
316         };
317
318         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
319
320         var balancedVariant = balancedVariantConverter.Convert(sequence);
321
322         // 1: [1]
323         // 2: [2]
324         // 3: [1,2]
325         // 4: [1,2,1,2]
326
327         var doublet = links.GetSource(balancedVariant);
328
329         var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
330
331         Assert.True(matchedSequences1.Count == 0);
332
333         var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
334
335         Assert.True(matchedSequences2.Count == 0);
336
337         var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
338
339         Assert.True(matchedSequences3.Count == 0);
340
341         var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
342
343         Assert.Contains(doublet, matchedSequences4);
344         Assert.Contains(balancedVariant, matchedSequences4);
345
346         for (var i = 0; i < sequence.Length; i++)
347         {
348             links.Delete(sequence[i]);
349         }
350     }
351 }
352
353 [Fact]
354 public static void IndexTest()
355 {
356     using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
357         ↪ true }, useSequences: true))
358     {
359         var links = scope.Links;
360         var sequences = scope.Sequences;
361         var index = sequences.Options.Index;
362
363         var e1 = links.Create();
364         var e2 = links.Create();
365
366         var sequence = new[]
367         {
368             e1, e2, e1, e2 // mama / papa
369         };
370
371         Assert.False(index.MightContain(sequence));
372
373         index.Add(sequence);
374
375         Assert.True(index.MightContain(sequence));
376     }
377 }
378
379 /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%
380 ↪ D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87
381 ↪ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>

```

```
private static readonly string _exampleText =  
    @"([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/Links  
↪ Platform/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)
```

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?

```
![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)
```

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?

```
![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)
```

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

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

```

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-anim
    ↳ ation-500.gif
    ↳ ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
    ↳ -animation-500.gif)";
433
434     private static readonly string _exampleLoremIpsumText =
435         @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
            ↳ incididunt ut labore et dolore magna aliqua.
436 Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
    ↳ consequat.";
437
438     [Fact]
439     public static void CompressionTest()
440     {
441         using (var scope = new TempLinksTestScope(useSequences: true))
442         {
443             var links = scope.Links;
444             var sequences = scope.Sequences;
445
446             var e1 = links.Create();
447             var e2 = links.Create();
448
449             var sequence = new[]
450             {
451                 e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
452             };
453
454             var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
455             var totalSequenceSymbolFrequencyCounter = new
                ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
456             var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
                ↳ totalSequenceSymbolFrequencyCounter);
457             var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
                ↳ balancedVariantConverter, doubletFrequenciesCache);
458

```

```

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

```

```

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

```

```

590 {
591     var sequence1 = compressed1[i];
592     var sequence2 = compressed2[i];
593     var sequence3 = compressed3[i];
594
595     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
596         ↳ scope1.Links.Unsync);
597
598     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
599         ↳ scope2.Links.Unsync);
600
601     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
602         ↳ scope3.Links.Unsync);
603
604     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
605         ↳ link.IsPartialPoint());
606     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
607         ↳ link.IsPartialPoint());
608     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
609         ↳ link.IsPartialPoint());
610
611     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
612     ↳ arrays[i].Length > 3)
613     //    Assert.False(structure1 == structure2);
614     //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
615     ↳ arrays[i].Length > 3)
616     //    Assert.False(structure3 == structure2);
617
618     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
619     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
620 }
621
622 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
623     ↳ totalCharacters);
624 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
625     ↳ totalCharacters);
626 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
627     ↳ totalCharacters);
628
629 Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
630     ↳ totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
631     ↳ totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
632     ↳ totalCharacters}");
633
634 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
635     ↳ scope2.Links.Unsync.Count() - initialCount2);
636 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
637     ↳ scope2.Links.Unsync.Count() - initialCount2);
638
639 var duplicateProvider1 = new
640     ↳ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
641 var duplicateProvider2 = new
642     ↳ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
643 var duplicateProvider3 = new
644     ↳ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
645
646 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
647 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
648 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
649
650 var duplicates1 = duplicateCounter1.Count();
651
652 ConsoleHelpers.Debug("-----");
653
654 var duplicates2 = duplicateCounter2.Count();
655
656 ConsoleHelpers.Debug("-----");
657
658 var duplicates3 = duplicateCounter3.Count();
659
660 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
661
662 linkFrequenciesCache1.ValidateFrequencies();
663 linkFrequenciesCache3.ValidateFrequencies();
664
665 }
666
667 }

```



```

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 }))
        using (var scope2 = new TempLinksTestScope(useSequences: true))
        {
669         scope1.Links.UseUnicode();
670         scope2.Links.UseUnicode();
671
672         //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
673         var compressor1 = scope1.Sequences;
674         var compressor2 = scope2.Sequences;
675
676         var compressed1 = new ulong[arrays.Length];
677         var compressed2 = new ulong[arrays.Length];
678
679         var sw1 = Stopwatch.StartNew();
680
681         var START = 0;
682         var END = arrays.Length;
683
684         // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
685         // Stability issue starts at 10001 or 11000
686         //for (int i = START; i < END; i++)
687         //{
688             // var first = compressor1.Compress(arrays[i]);
689             // var second = compressor1.Compress(arrays[i]);
690
691             // if (first == second)
692             //     compressed1[i] = first;
693             // else
694             // {
695                 // TODO: Find a solution for this case
696             // }
697         //}
698
699         for (int i = START; i < END; i++)
700         {
701             var first = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
702             var second = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
703
704             if (first == second)
705             {
706                 compressed1[i] = first;
707             }
708             else
709             {
710                 // TODO: Find a solution for this case
711             }
712         }
713
714         var elapsed1 = sw1.Elapsed;
715
716         var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
717
718         var sw2 = Stopwatch.StartNew();
719
720         for (int i = START; i < END; i++)
721         {
722             var first = balancedVariantConverter.Convert(arrays[i]);
723             var second = balancedVariantConverter.Convert(arrays[i]);
724
725
726

```

```

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

```

```

797 using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
    ↳ SequencesOptions<ulong> { UseCompression = true,
    ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
798 using (var scope2 = new TempLinksTestScope(useSequences: true))
799 {
800     scope1.Links.UseUnicode();
801     scope2.Links.UseUnicode();
802
803     var compressor1 = scope1.Sequences;
804     var compressor2 = scope2.Sequences;
805
806     var compressed1 = new ulong[arrays.Length];
807     var compressed2 = new ulong[arrays.Length];
808
809     var sw1 = Stopwatch.StartNew();
810
811     var START = 0;
812     var END = arrays.Length;
813
814     for (int i = START; i < END; i++)
815     {
816         compressed1[i] = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
817     }
818
819     var elapsed1 = sw1.Elapsed;
820
821     var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
822
823     var sw2 = Stopwatch.StartNew();
824
825     for (int i = START; i < END; i++)
826     {
827         compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
828     }
829
830     var elapsed2 = sw2.Elapsed;
831
832     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
    ↳ {elapsed2}");
833
834     Assert.True(elapsed1 > elapsed2);
835
836     // Checks
837     for (int i = START; i < END; i++)
838     {
839         var sequence1 = compressed1[i];
840         var sequence2 = compressed2[i];
841
842         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
843         {
844             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
    ↳ scope1.Links);
845
846             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
    ↳ scope2.Links);
847
848             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
849         }
850     }
851
852     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
853     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
854
855     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
    ↳ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
    ↳ totalCharacters}}");
856
857     // Can be worse than balanced variant
858     //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
859
860     //compressor1.ValidateFrequencies();
861 }
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
886     Global.Trash = createResults;
887
888     for (var i = 0; i < sequenceLength; i++)
889     {
890         links.Delete(sequence[i]);
891     }
892 }
893
894 [Fact]
895 public static void AllPossibleConnectionsTest()
896 {
897     const long sequenceLength = 5;
898
899     using (var scope = new TempLinksTestScope(useSequences: true))
900     {
901         var links = scope.Links;
902         var sequences = scope.Sequences;
903
904         var sequence = new ulong[sequenceLength];
905         for (var i = 0; i < sequenceLength; i++)
906         {
907             sequence[i] = links.Create();
908         }
909
910         var createResults = sequences.CreateAllVariants2(sequence);
911         var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
912
913         for (var i = 0; i < 1; i++)
914         {
915             var sw1 = Stopwatch.StartNew();
916             var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
917
918             var sw2 = Stopwatch.StartNew();
919             var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
920
921             var sw3 = Stopwatch.StartNew();
922             var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
923
924             var sw4 = Stopwatch.StartNew();
925             var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
926
927             Global.Trash = searchResults3;
928             Global.Trash = searchResults4; //-V3008
929
930             var intersection1 = createResults.Intersect(searchResults1).ToList();
931             Assert.True(intersection1.Count == createResults.Length);
932
933             var intersection2 = reverseResults.Intersect(searchResults1).ToList();
934             Assert.True(intersection2.Count == reverseResults.Length);
935
936             var intersection0 = searchResults1.Intersect(searchResults2).ToList();
937             Assert.True(intersection0.Count == searchResults2.Count);
938
939             var intersection3 = searchResults2.Intersect(searchResults3).ToList();
940             Assert.True(intersection3.Count == searchResults3.Count);
941
942             var intersection4 = searchResults3.Intersect(searchResults4).ToList();
943             Assert.True(intersection4.Count == searchResults4.Count);
944         }
945
946         for (var i = 0; i < sequenceLength; i++)
947         {
948

```

```

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
19         ↪ useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
20         ↪ useLog) { }
21
22         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
23         ↪ true, bool useSequences = false, bool useLog = false)
24         {
25             _deleteFiles = deleteFiles;
26             TempFilename = Path.GetTempFileName();
27             TempTransactionLogFilename = Path.GetTempFileName();
28             var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);

```

```

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

```

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

```

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

```

```

50 // Update link to reference null (prepare for delete)
51 var updated = links.Update(linkAddress, constants.Null, constants.Null);
52
53 Assert.True(equalityComparer.Equals(updated, linkAddress));
54
55 link = new Link<T>(links.GetLink(linkAddress));
56
57 Assert.True(equalityComparer.Equals(link.Source, constants.Null));
58 Assert.True(equalityComparer.Equals(link.Target, constants.Null));
59
60 // Delete link
61 links.Delete(linkAddress);
62
63 Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
64
65 setter = new Setter<T>(constants.Null);
66 links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
67
68 Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
69 }
70
71 public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
72 {
73     // Constants
74     var constants = links.Constants;
75     var equalityComparer = EqualityComparer<T>.Default;
76
77     var h106E = new Hybrid<T>(106L, isExternal: true);
78     var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
79     var h108E = new Hybrid<T>(-108L);
80
81     Assert.Equal(106L, h106E.AbsoluteValue);
82     Assert.Equal(107L, h107E.AbsoluteValue);
83     Assert.Equal(108L, h108E.AbsoluteValue);
84
85     // Create Link (External -> External)
86     var linkAddress1 = links.Create();
87
88     links.Update(linkAddress1, h106E, h108E);
89
90     var link1 = new Link<T>(links.GetLink(linkAddress1));
91
92     Assert.True(equalityComparer.Equals(link1.Source, h106E));
93     Assert.True(equalityComparer.Equals(link1.Target, h108E));
94
95     // Create Link (Internal -> External)
96     var linkAddress2 = links.Create();
97
98     links.Update(linkAddress2, linkAddress1, h108E);
99
100     var link2 = new Link<T>(links.GetLink(linkAddress2));
101
102     Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
103     Assert.True(equalityComparer.Equals(link2.Target, h108E));
104
105     // Create Link (Internal -> Internal)
106     var linkAddress3 = links.Create();
107
108     links.Update(linkAddress3, linkAddress1, linkAddress2);
109
110     var link3 = new Link<T>(links.GetLink(linkAddress3));
111
112     Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
113     Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
114
115     // Search for created link
116     var setter1 = new Setter<T>(constants.Null);
117     links.Each(h106E, h108E, setter1.SetAndReturnFalse);
118
119     Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
120
121     // Search for nonexistent link
122     var setter2 = new Setter<T>(constants.Null);
123     links.Each(h106E, h107E, setter2.SetAndReturnFalse);
124
125     Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
126
127     // Update link to reference null (prepare for delete)
128     var updated = links.Update(linkAddress3, constants.Null, constants.Null);
129

```

```

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

```

./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                 new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeleti
41                 ↪ ons(100);
42             }
43
44             [Fact]
45             public static void CascadeUpdateTest()
46             {
47                 var itself = _constants.Itself;
48
49                 using (var scope = new TempLinksTestScope(useLog: true))
50                 {
51                     var links = scope.Links;
52
53                     var l1 = links.Create();
54                     var l2 = links.Create();
55
56                     l2 = links.Update(l2, l2, l1, l2);
57
58                     links.CreateAndUpdate(l2, itself);
59                     links.CreateAndUpdate(l2, itself);
60
61                     l2 = links.Update(l2, l1);
62
63                     links.Delete(l2);
64
65                     Global.Trash = links.Count();
66
67                     links.Unsync.DisposeIfPossible(); // Close links to access log
68
69                     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
70                     ↪ e.TempTransactionLogFilename);
71                 }
72
73                 [Fact]
74                 public static void BasicTransactionLogTest()
75                 {
76                     using (var scope = new TempLinksTestScope(useLog: true))
77                     {
78                         var links = scope.Links;
79                         var l1 = links.Create();
80                         var l2 = links.Create();
81
82                         Global.Trash = links.Update(l2, l2, l1, l2);
83
84                         links.Delete(l1);
85
86                         links.Unsync.DisposeIfPossible(); // Close links to access log
87
88                         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
89                         ↪ e.TempTransactionLogFilename);
90                     }
91                 }
92             }
93         }
94     }
95 }

```

```

89 [Fact]
90 public static void TransactionAutoRevertedTest()
91 {
92     // Auto Reverted (Because no commit at transaction)
93     using (var scope = new TempLinksTestScope(useLog: true))
94     {
95         var links = scope.Links;
96         var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
97         using (var transaction = transactionsLayer.BeginTransaction())
98         {
99             var l1 = links.Create();
100             var l2 = links.Create();
101
102             links.Update(l2, l2, l1, l2);
103         }
104
105         Assert.Equal(0UL, links.Count());
106
107         links.Unsync.DisposeIfPossible();
108
109         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
110         Assert.Single(transitions);
111     }
112 }
113
114 [Fact]
115 public static void TransactionUserCodeErrorNoDataSavedTest()
116 {
117     // User Code Error (Autoreverted), no data saved
118     var itself = _constants.Itself;
119
120     TempLinksTestScope lastScope = null;
121     try
122     {
123         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
124             useLog: true))
125         {
126             var links = scope.Links;
127             var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecorator)
128                 atorBase<ulong>)links.Unsync).Links;
129             using (var transaction = transactionsLayer.BeginTransaction())
130             {
131                 var l1 = links.CreateAndUpdate(itself, itself);
132                 var l2 = links.CreateAndUpdate(itself, itself);
133
134                 l2 = links.Update(l2, l2, l1, l2);
135
136                 links.CreateAndUpdate(l2, itself);
137                 links.CreateAndUpdate(l2, itself);
138
139                 //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
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>(lastScope.TempTransactionLogFilename);
157
158         Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
159             transitions[0].After.IsNull());
160
161         lastScope.DeleteFiles();
162     }
163 }

```

```

162 }
163
164 [Fact]
165 public static void TransactionUserCodeErrorSomeDataSavedTest()
166 {
167     // User Code Error (Autoreverted), some data saved
168     var itself = _constants.Itself;
169
170     TempLinksTestScope lastScope = null;
171     try
172     {
173         ulong l1;
174         ulong l2;
175
176         using (var scope = new TempLinksTestScope(useLog: true))
177         {
178             var links = scope.Links;
179             l1 = links.CreateAndUpdate(itself, itself);
180             l2 = links.CreateAndUpdate(itself, itself);
181
182             l2 = links.Update(l2, l2, l1, l2);
183
184             links.CreateAndUpdate(l2, itself);
185             links.CreateAndUpdate(l2, itself);
186
187             links.Unsync.DisposeIfPossible();
188
189             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
190                 ↪ scope.TempTransactionLogFilename);
191         }
192
193         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
194             ↪ useLog: true))
195         {
196             var links = scope.Links;
197             var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
198             using (var transaction = transactionsLayer.BeginTransaction())
199             {
200                 l2 = links.Update(l2, l1);
201
202                 links.Delete(l2);
203
204                 ExceptionThrower();
205
206                 transaction.Commit();
207             }
208
209             Global.Trash = links.Count();
210         }
211     }
212     catch
213     {
214         Assert.False(lastScope == null);
215
216         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
217             ↪ Scope.TempTransactionLogFilename);
218
219         lastScope.DeleteFiles();
220     }
221 }
222
223 [Fact]
224 public static void TransactionCommit()
225 {
226     var itself = _constants.Itself;
227
228     var tempDatabaseFilename = Path.GetTempFileName();
229     var tempTransactionLogFilename = Path.GetTempFileName();
230
231     // Commit
232     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
233         ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
234         ↪ tempTransactionLogFilename))
235     using (var links = new UInt64Links(memoryAdapter))
236     {
237         using (var transaction = memoryAdapter.BeginTransaction())
238         {
239             var l1 = links.CreateAndUpdate(itself, itself);
240             var l2 = links.CreateAndUpdate(itself, itself);
241         }
242     }
243 }

```

```

236         Global.Trash = links.Update(l2, l2, l1, l2);
237
238         links.Delete(l1);
239
240         transaction.Commit();
241     }
242
243     Global.Trash = links.Count();
244 }
245
246 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
    ↪ sactionLogFilename);
247
248 }
249
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
283     // Try load damaged database
284     try
285     {
286         // TODO: Fix
287         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
    ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
    ↪ tempTransactionLogFilename))
288         using (var links = new UInt64Links(memoryAdapter))
289         {
290             Global.Trash = links.Count();
291         }
292     }
293     catch (NotSupportedException ex)
294     {
295         Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
    ↪ yet.");
296     }
297
298     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
    ↪ sactionLogFilename);
299
300     File.Delete(tempDatabaseFilename);
301     File.Delete(tempTransactionLogFilename);
302 }
303
304 [Fact]
305 public static void Bug1Test()

```

```

306 {
307     var tempDatabaseFilename = Path.GetTempFileName();
308     var tempTransactionLogFilename = Path.GetTempFileName();
309
310     var itself = _constants.Itself;
311
312     // User Code Error (Autoreverted), some data saved
313     try
314     {
315         ulong l1;
316         ulong l2;
317
318         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
319             ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
320             ↳ tempTransactionLogFilename))
321         using (var links = new UInt64Links(memoryAdapter))
322         {
323             l1 = links.CreateAndUpdate(itself, itself);
324             l2 = links.CreateAndUpdate(itself, itself);
325
326             l2 = links.Update(l2, l2, l1, l2);
327
328             links.CreateAndUpdate(l2, itself);
329             links.CreateAndUpdate(l2, itself);
330         }
331
332         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
333             ↳ TransactionLogFilename);
334
335         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
336             ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
337             ↳ tempTransactionLogFilename))
338         using (var links = new UInt64Links(memoryAdapter))
339         {
340             using (var transaction = memoryAdapter.BeginTransaction())
341             {
342                 l2 = links.Update(l2, l1);
343
344                 links.Delete(l2);
345
346                 ExceptionThrower();
347
348                 transaction.Commit();
349             }
350
351             Global.Trash = links.Count();
352         }
353     }
354     catch
355     {
356         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
357             ↳ TransactionLogFilename);
358     }
359
360     File.Delete(tempDatabaseFilename);
361     File.Delete(tempTransactionLogFilename);
362 }
363
364 private static void ExceptionThrower() => throw new InvalidOperationException();
365
366 [Fact]
367 public static void PathsTest()
368 {
369     var source = _constants.SourcePart;
370     var target = _constants.TargetPart;
371
372     using (var scope = new TempLinksTestScope())
373     {
374         var links = scope.Links;
375         var l1 = links.CreatePoint();
376         var l2 = links.CreatePoint();
377
378         var r1 = links.GetByKeys(l1, source, target, source);
379         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
380     }
381 }
382
383 [Fact]
384 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
428
429 /*
430 public static void RunAllPerformanceTests()
431 {
432     try
433     {
434         links.TestLinksInSteps();
435     }
436     catch (Exception ex)
437     {
438         ex.WriteToConsole();
439     }
440
441     return;
442
443     try
444     {
445         //ThreadPool.SetMaxThreads(2, 2);
446
447         // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
448         ↳ результат
449         // Также это дополнительно помогает в отладке
450         // Увеличивает вероятность попадания информации в кэши
451         for (var i = 0; i < 10; i++)
452         {
453             //0 - 10 ГБ
454             //Каждые 100 МБ срез цифр
455
456             //links.TestGetSourceFunction();
457             //links.TestGetSourceFunctionInParallel();
458         }
459     }
460 }

```

```

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.Write("\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.Write("\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}",
583                 Iterations, elapsedTime, (long)iterationsPerSecond, counter);
584         }
585     }
586
587     [Fact(Skip = "performance test")]
588     public static void GetSourceInParallel()
589     {
590         using (var scope = new TempLinksTestScope())
591         {
592             var links = scope.Links;
593             ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
↪ parallel.", Iterations);
594
595             long counter = 0;
596
597             //var firstLink = links.First();
598             var firstLink = links.Create();

```



```

599
600     var sw = Stopwatch.StartNew();
601
602     // Тестируем саму функцию
603     Parallel.For(0, Iterations, x =>
604     {
605         Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
606         //Interlocked.Increment(ref counter);
607     });
608
609     var elapsedTime = sw.Elapsed;
610
611     var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
612
613     links.Delete(firstLink);
614
615     ConsoleHelpers.Debug(
616         "{0} Iterations of GetSource function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
617     }
618 }
619
620
621 [Fact(Skip = "performance test")]
622 public static void TestGetTarget()
623 {
624     using (var scope = new TempLinksTestScope())
625     {
626         var links = scope.Links;
627         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
        ↳ Iterations);
628
629         ulong counter = 0;
630
631         //var firstLink = links.First();
632         var firstLink = links.Create();
633
634         var sw = Stopwatch.StartNew();
635
636         for (ulong i = 0; i < Iterations; i++)
637         {
638             counter += links.GetTarget(firstLink);
639         }
640
641         var elapsedTime = sw.Elapsed;
642
643         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
644
645         links.Delete(firstLink);
646
647         ConsoleHelpers.Debug(
648             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
649     }
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         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
676
677         links.Delete(firstLink);
678
679         ConsoleHelpers.Debug(
680             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
681             ↪ 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
725     File.Delete(tempFilename);
726 }*/
727
728 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
729 public static void TestRandomSearchAll()
730 {
731     using (var scope = new TempLinksTestScope())
732     {
733         var links = scope.Links;
734         ulong counter = 0;
735
736         var maxLink = links.Count();
737
738         var iterations = links.Count();
739
740         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
741 ↪ links.Count());
742
743         var sw = Stopwatch.StartNew();
744
745         for (var i = iterations; i > 0; i--)
746         {
747             var linksAddressRange = new
748 ↪ Range<ulong>(_constants.PossibleInnerReferencesRange.Minimum, maxLink);
749
750             var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
751             var target = RandomHelpers.Default.NextUInt64(linksAddressRange);

```

```

745         counter += links.SearchOrDefault(source, target);
746     }
747
748     var elapsedTime = sw.Elapsed;
749
750     var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
751
752     ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪     Iterations per second), c: {3}",
        iterations, elapsedTime, (long)iterationsPerSecond, counter);
753
754     }
755 }
756
757 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
758 public static void TestEach()
759 {
760     using (var scope = new TempLinksTestScope())
761     {
762         var links = scope.Links;
763
764         var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
765
766         ConsoleHelpers.Debug("Testing Each function.");
767
768         var sw = Stopwatch.StartNew();
769
770         links.Each(counter.IncrementAndReturnTrue);
771
772         var elapsedTime = sw.Elapsed;
773
774         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
775
776         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
↪     links per second)",
            counter, elapsedTime, (long)linksPerSecond);
777
778     }
779 }
780
781 /*
782 [Fact]
783 public static void TestForeach()
784 {
785     var tempFilename = Path.GetTempFileName();
786
787     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪     DefaultLinksSizeStep))
788     {
789         ulong counter = 0;
790
791         ConsoleHelpers.Debug("Testing foreach through links.");
792
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

```

```

820     long counter = 0;
821
822     ConsoleHelpers.Debug("Testing parallel foreach through links.");
823
824     var sw = Stopwatch.StartNew();
825
826     //Parallel.ForEach((IEnumerable<ulong>)links, x =>
827     //{
828     //    Interlocked.Increment(ref counter);
829     //});
830
831     var elapsedTime = sw.Elapsed;
832
833     var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
834
835     ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↪ {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
836 }
837
838 File.Delete(tempFilename);
839 }
840 */
841
842 [Fact(Skip = "performance test")]
843 public static void Create64BillionLinks()
844 {
845     using (var scope = new TempLinksTestScope())
846     {
847         var links = scope.Links;
848         var linksBeforeTest = links.Count();
849
850         long linksToCreate = 64 * 1024 * 1024 /
↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
851
852         ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
853
854         var elapsedTime = Performance.Measure(() =>
855         {
856             for (long i = 0; i < linksToCreate; i++)
857             {
858                 links.Create();
859             }
860         });
861
862         var linksCreated = links.Count() - linksBeforeTest;
863         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
864
865         ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
866
867         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
↪ linksCreated, elapsedTime,
            (long)linksPerSecond);
868     }
869 }
870
871 [Fact(Skip = "performance test")]
872 public static void Create64BillionLinksInParallel()
873 {
874     using (var scope = new TempLinksTestScope())
875     {
876         var links = scope.Links;
877         var linksBeforeTest = links.Count();
878
879         var sw = Stopwatch.StartNew();
880
881         long linksToCreate = 64 * 1024 * 1024 /
↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
882
883         ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
884
885         Parallel.For(0, linksToCreate, x => links.Create());
886
887         var elapsedTime = sw.Elapsed;
888
889         var linksCreated = links.Count() - linksBeforeTest;
890         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
891
892         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
↪ linksCreated, elapsedTime,
            (long)linksPerSecond);
893     }
894 }

```

```

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

```

./Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs

```

1  using Xunit;
2  using Platform.Random;
3  using Platform.Data.Doublets.Numbers.Unary;
4
5  namespace Platform.Data.Doublets.Tests
6  {
7      public static class UnaryNumberConvertersTests
8      {
9          [Fact]
10         public static void ConvertersTest()
11         {
12             using (var scope = new TempLinksTestScope())
13             {
14                 const int N = 10;
15                 var links = scope.Links;
16                 var meaningRoot = links.CreatePoint();
17                 var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
18                 var powerOf2ToUnaryNumberConverter = new
19                     ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
20                 var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
21                     ↪ powerOf2ToUnaryNumberConverter);
22                 var random = new System.Random(0);
23                 ulong[] numbers = new ulong[N];
24                 ulong[] unaryNumbers = new ulong[N];
25                 for (int i = 0; i < N; i++)
26                 {
27                     numbers[i] = random.NextUInt64();
28                     unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
29                 }
30                 var fromUnaryNumberConverterUsingOrOperation = new
31                     ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
32                     ↪ powerOf2ToUnaryNumberConverter);
33                 var fromUnaryNumberConverterUsingAddOperation = new
34                     ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
35                 for (int i = 0; i < N; i++)
36                 {
37                     Assert.Equal(numbers[i],
38                         ↪ fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
39                     Assert.Equal(numbers[i],
40                         ↪ fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
41                 }
42             }
43         }
44     }
45 }

```

./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         private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
56             ↪ meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
57             ↪ numberToAddressConverter)
58         {
59             var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
60             var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
61                 ↪ addressToNumberConverter, unicodeSymbolMarker);
62             var originalCharacter = 'H';
63             var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
64             var unicodeSymbolCriterionMatcher = new UnicodeSymbolCriterionMatcher<ulong>(links,
65                 ↪ unicodeSymbolMarker);
66             var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
67                 ↪ numberToAddressConverter, unicodeSymbolCriterionMatcher);
68             var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
69             Assert.Equal(originalCharacter, resultingCharacter);
70         }
71
72         [Fact]
73         public static void StringAndUnicodeSequenceConvertersTest()
74         {
75             using (var scope = new TempLinksTestScope())
76             {
77                 var links = scope.Links;
78
79                 var itself = links.Constants.Itself;
80
81                 var meaningRoot = links.CreatePoint();

```

```

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

```

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, 148
- ./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs, 149
- ./Platform.Data.Doublets.Tests/ScopeTests.cs, 150
- ./Platform.Data.Doublets.Tests/SequencesTests.cs, 151
- ./Platform.Data.Doublets.Tests/TempLinksTestScope.cs, 165
- ./Platform.Data.Doublets.Tests/TestExtensions.cs, 166
- ./Platform.Data.Doublets.Tests/UInt64LinksTests.cs, 168
- ./Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs, 181
- ./Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs, 181
- ./Platform.Data.Doublets/Decorators/LinksCascadeUniquenessAndUsagesResolver.cs, 1
- ./Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.cs, 1
- ./Platform.Data.Doublets/Decorators/LinksDecoratorBase.cs, 1
- ./Platform.Data.Doublets/Decorators/LinksDisposableDecoratorBase.cs, 2
- ./Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.cs, 3
- ./Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.cs, 3
- ./Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.cs, 4
- ./Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.cs, 4
- ./Platform.Data.Doublets/Decorators/LinksUniquenessResolver.cs, 5
- ./Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs, 5
- ./Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs, 5
- ./Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs, 6
- ./Platform.Data.Doublets/Decorators/UInt64Links.cs, 6
- ./Platform.Data.Doublets/Decorators/UniLinks.cs, 7
- ./Platform.Data.Doublets/Doublet.cs, 12
- ./Platform.Data.Doublets/DoubletComparer.cs, 12
- ./Platform.Data.Doublets/Hybrid.cs, 13
- ./Platform.Data.Doublets/ILinks.cs, 14
- ./Platform.Data.Doublets/ILinksExtensions.cs, 15
- ./Platform.Data.Doublets/ISynchronizedLinks.cs, 26
- ./Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs, 25
- ./Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs, 26
- ./Platform.Data.Doublets/Link.cs, 26
- ./Platform.Data.Doublets/LinkExtensions.cs, 29
- ./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, 31
- ./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, 45
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinks.cs, 53
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinksBase.cs, 46
- ./Platform.Data.Doublets/ResizableDirectMemory/Generic/UnusedLinksListMethods.cs, 54
- ./Platform.Data.Doublets/ResizableDirectMemory/ILinksListMethods.cs, 55
- ./Platform.Data.Doublets/ResizableDirectMemory/ILinksTreeMethods.cs, 55
- ./Platform.Data.Doublets/ResizableDirectMemory/LinksHeader.cs, 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, 60
- ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs, 61

./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs, 62
./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64ResizableDirectMemoryLinks.cs, 63
./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64UnusedLinksListMethods.cs, 65
./Platform.Data.Doublets/Sequences/ArrayExtensions.cs, 65
./Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 66
./Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 66
./Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 69
./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 70
./Platform.Data.Doublets/Sequences/Converters/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, 75
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 77
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToltsFrequencyValueConverter.cs, 77
./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 78
./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 78
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 79
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 79
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 79
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 80
./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 81
./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 81
./Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 82
./Platform.Data.Doublets/Sequences/IListExtensions.cs, 82
./Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 82
./Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 83
./Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 84
./Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 84
./Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 84
./Platform.Data.Doublets/Sequences/Indexes/Unindex.cs, 85
./Platform.Data.Doublets/Sequences/ListFiller.cs, 85
./Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 96
./Platform.Data.Doublets/Sequences/Sequences.cs, 86
./Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 122
./Platform.Data.Doublets/Sequences/SequencesOptions.cs, 123
./Platform.Data.Doublets/Sequences/SetFiller.cs, 124
./Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 125
./Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 125
./Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 126
./Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 127
./Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 128
./Platform.Data.Doublets/Stacks/Stack.cs, 129
./Platform.Data.Doublets/Stacks/StackExtensions.cs, 129
./Platform.Data.Doublets/SynchronizedLinks.cs, 130
./Platform.Data.Doublets/UInt64LinksExtensions.cs, 130
./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 133
./Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 138
./Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 138
./Platform.Data.Doublets/Unicode/UnicodeMap.cs, 139
./Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs, 141
./Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 141
./Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs, 142
./Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 142