

LinksPlatform's Platform.Data.Doublets Class Library

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

```
1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Decorators
6 {
7     public class LinksCascadeUniquenessAndUsagesResolver<TLink> : LinksUniquenessResolver<TLink>
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public LinksCascadeUniquenessAndUsagesResolver(ILinks<TLink> links) : base(links) { }
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         protected override TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
14             ↪ newLinkAddress)
15         {
16             // Use Facade (the last decorator) to ensure recursion working correctly
17             _facade.MergeUsages(oldLinkAddress, newLinkAddress);
18             return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
19         }
20     }
```

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

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

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

```
1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Decorators
8 {
9     public abstract class LinksDecoratorBase<TLink> : LinksOperatorBase<TLink>, ILinks<TLink>
10     {
11         protected readonly LinksConstants<TLink> _constants;
12
13         public LinksConstants<TLink> Constants
14         {
15             [MethodImpl(MethodImplOptions.AggressiveInlining)]
16             get => _constants;
17         }
18
19         protected ILinks<TLink> _facade;
20
21         public ILinks<TLink> Facade
22         {
23             [MethodImpl(MethodImplOptions.AggressiveInlining)]
24             get => _facade;
25             [MethodImpl(MethodImplOptions.AggressiveInlining)]
26             set
27             {
28             }
```

```

28         _facade = value;
29         if (_links is LinksDecoratorBase<TLink> decorator)
30         {
31             decorator.Facade = value;
32         }
33     }
34 }
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 protected LinksDecoratorBase(ILinks<TLink> links) : base(links)
38 {
39     _constants = links.Constants;
40     Facade = this;
41 }
42
43 [MethodImpl(MethodImplOptions.AggressiveInlining)]
44 public virtual TLink Count(IList<TLink> restrictions) => _links.Count(restrictions);
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
48     => _links.Each(handler, restrictions);
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 public virtual TLink Create(IList<TLink> restrictions) => _links.Create(restrictions);
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
55     _links.Update(restrictions, substitution);
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 public virtual void Delete(IList<TLink> restrictions) => _links.Delete(restrictions);
59 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Disposables;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5 #pragma warning disable CA1063 // Implement IDisposable Correctly
6
7 namespace Platform.Data.Doublets.Decorators
8 {
9     public abstract class LinksDisposableDecoratorBase<TLink> : LinksDecoratorBase<TLink>,
10         ↳ ILinks<TLink>, System.IDisposable
11     {
12         protected class DisposableWithMultipleCallsAllowed : Disposable
13         {
14             [MethodImpl(MethodImplOptions.AggressiveInlining)]
15             public DisposableWithMultipleCallsAllowed(Disposal disposal) : base(disposal) { }
16
17             protected override bool AllowMultipleDisposeCalls
18             {
19                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
20                 get => true;
21             }
22         }
23
24         protected readonly DisposableWithMultipleCallsAllowed Disposable;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected LinksDisposableDecoratorBase(ILinks<TLink> links) : base(links) => Disposable
28             ↳ = new DisposableWithMultipleCallsAllowed(Dispose);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         ~LinksDisposableDecoratorBase() => Disposable.Destruct();
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public void Dispose() => Disposable.Dispose();
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected virtual void Dispose(bool manual, bool wasDisposed)
38         {
39             if (!wasDisposed)
40             {
41                 _links.DisposeIfPossible();
42             }
43         }
44     }
45 }

```

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

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Decorators
8  {
9      public class LinksItselfConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12         ↪ EqualityComparer<TLink>.Default;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public LinksItselfConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
19         {
20             var constants = _constants;
21             var itselfConstant = constants.Itself;
22             if (!_equalityComparer.Equals(constants.Any, itselfConstant) &&
23             ↪ restrictions.Contains(itselfConstant))
24             {
25                 // Itself constant is not supported for Each method right now, skipping execution
26                 return constants.Continue;
27             }
28             return _links.Each(handler, restrictions);
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
33         ↪ _links.Update(restrictions, _links.ResolveConstantAsSelfReference(_constants.Itself,
34         ↪ restrictions, substitution));

```

```

31     }
32 }

```

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

```

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

```

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

```

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

```

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

```

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

```

```

20     var newLinkAddress = links.SearchOrDefault(substitution[constants.SourcePart],
21         ↪ substitution[constants.TargetPart]);
22     if (_equalityComparer.Equals(newLinkAddress, default))
23     {
24         return links.Update(restrictions, substitution);
25     }
26     return ResolveAddressChangeConflict(restrictions[constants.IndexPart],
27         ↪ newLinkAddress);
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 }

```

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

```

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

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      public class LinksUsagesValidator<TLink> : LinksDecoratorBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksUsagesValidator(ILinks<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
15         {
16             var links = _links;
17             links.EnsureNoUsages(restrictions[_constants.IndexPart]);
18             return links.Update(restrictions, substitution);
19         }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public override void Delete(IList<TLink> restrictions)
23         {
24             var link = restrictions[_constants.IndexPart];
25             var links = _links;
26             links.EnsureNoUsages(link);
27             links.Delete(link);
28         }
29     }
30 }

```

```

29     }
30 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Decorators
7 {
8     public class NonNullContentsLinkDeletionResolver<TLink> : LinksDecoratorBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public NonNullContentsLinkDeletionResolver(ILinks<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public override void Delete(IList<TLink> restrictions)
15         {
16             var linkIndex = restrictions[_constants.IndexPart];
17             var links = _links;
18             links.EnforceResetValues(linkIndex);
19             links.Delete(linkIndex);
20         }
21     }
22 }

```

1.13 ./csharp/Platform.Data.Doublets/Decorators/UInt64Links.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Decorators
7 {
8     /// <summary>
9     /// <para>Represents a combined decorator that implements the basic logic for interacting
10     ///   ↪ with the links storage for links with addresses represented as <see cref="System.UInt64">
11     ///   ↪ </para>
12     /// <para>Представляет комбинированный декоратор, реализующий основную логику по
13     ///   ↪ взаимодействию с хранилищем связей, для связей с адресами представленными в виде <see
14     ///   ↪ cref="System.UInt64"/>.</para>
15     /// </summary>
16     /// <remarks>
17     /// Возможные оптимизации:
18     /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
19     ///   + меньше объём БД
20     ///   - меньше производительность
21     ///   - больше ограничение на количество связей в БД)
22     /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
23     ///   + меньше объём БД
24     ///   - больше сложность
25     ///
26     /// Текущее теоретическое ограничение на индекс связи, из-за использования 5 бит в размере
27     ///   ↪ поддеревьев для AVL баланса и флагов нитей: 2 в степени(64 минус 5 равно 59 ) равно 576
28     ///   ↪ 460 752 303 423 488
29     /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
30     ///   ↪ (битовыми строками) - вариант матрицы (выстраиваемой лениво).
31     ///
32     /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
33     ///   ↪ выбрасываться только при #if DEBUG
34     /// </remarks>
35     public class UInt64Links : LinksDisposableDecoratorBase<ulong>
36     {
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public UInt64Links(ILinks<ulong> links) : base(links) { }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public override ulong Create(IList<ulong> restrictions) => _links.CreatePoint();
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
45         {
46             var constants = _constants;
47             var indexPartConstant = constants.IndexPart;
48             var sourcePartConstant = constants.SourcePart;
49             var targetPartConstant = constants.TargetPart;
50             var nullConstant = constants.Null;
51             var itselfConstant = constants.Itself;

```

```

44     var existedLink = nullConstant;
45     var updatedLink = restrictions[indexPartConstant];
46     var newSource = substitution[sourcePartConstant];
47     var newTarget = substitution[targetPartConstant];
48     var links = _links;
49     if (newSource != itselfConstant && newTarget != itselfConstant)
50     {
51         existedLink = links.SearchOrDefault(newSource, newTarget);
52     }
53     if (existedLink == nullConstant)
54     {
55         var before = links.GetLink(updatedLink);
56         if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
57             ↪ newTarget)
58         {
59             links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
60                 ↪ newSource,
61                                     newTarget == itselfConstant ? updatedLink :
62                 ↪ newTarget);
63         }
64         return updatedLink;
65     }
66     else
67     {
68         return _facade.MergeAndDelete(updatedLink, existedLink);
69     }
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 public override void Delete(ICollection<ulong> restrictions)
74 {
75     var linkIndex = restrictions[_constants.IndexPart];
76     var links = _links;
77     links.EnforceResetValues(linkIndex);
78     _facade.DeleteAllUsages(linkIndex);
79     links.Delete(linkIndex);
80 }
81 }
82 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using Platform.Collections;
5  using Platform.Collections.Lists;
6  using Platform.Data.Universal;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Decorators
11 {
12     /// <remarks>
13     /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
14     /// Now we go with nothing. And nothing is something one, but empty, and cannot be changed
15     ↪ by itself. But can cause creation (update from nothing) or deletion (update to nothing).
16     ///
17     /// TODO: Decide to change to IDoubletLinks or not to change. (Better to create
18     ↪ DefaultUniLinksBase, that contains logic itself and can be implemented using both
19     ↪ IDoubletLinks and ILinks.)
20     /// </remarks>
21     internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
22     {
23         private static readonly EqualityComparer<TLink> _equalityComparer =
24             ↪ EqualityComparer<TLink>.Default;
25
26         public UniLinks(ILinks<TLink> links) : base(links) { }
27
28         private struct Transition
29         {
30             public IList<TLink> Before;
31             public IList<TLink> After;
32
33             public Transition(IList<TLink> before, IList<TLink> after)
34             {
35                 Before = before;
36                 After = after;
37             }
38         }
39     }
40 }

```

```

36 //public static readonly TLink NullConstant = Use<LinksConstants<TLink>>.Single.Null;
37 //public static readonly IReadOnlyList<TLink> NullLink = new
    ↳ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
    ↳ });
38
39 // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
    ↳ (Links-Expression)
40 public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
    ↳ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
    ↳ substitutedHandler)
41 {
42     /////List<Transition> transitions = null;
43     /////if (!restriction.IsNullOrEmpty())
44     /////{
45     /////    // Есть причина делать проход (чтение)
46     /////    if (matchedHandler != null)
47     /////    {
48     /////        if (!substitution.IsNullOrEmpty())
49     /////        {
50     /////            // restriction => { 0, 0, 0 } | { 0 } // Create
51     /////            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
    ↳ Create / Update
52     /////            // substitution => { 0, 0, 0 } | { 0 } // Delete
53     /////            transitions = new List<Transition>();
54     /////            if (Equals(substitution[Constants.IndexPart], Constants.Null))
55     /////            {
56     /////                // If index is Null, that means we always ignore every other
    ↳ value (they are also Null by definition)
57     /////                var matchDecision = matchedHandler(, NullLink);
58     /////                if (Equals(matchDecision, Constants.Break))
59     /////                {
60     /////                    return false;
61     /////                }
62     /////                if (!Equals(matchDecision, Constants.Skip))
63     /////                {
64     /////                    transitions.Add(new Transition(matchedLink, newValue));
65     /////                }
66     /////            }
67     /////            else
68     /////            {
69     /////                Func<T, bool> handler;
70     /////                handler = link =>
71     /////                {
72     /////                    var matchedLink = Memory.GetLinkValue(link);
73     /////                    var newValue = Memory.GetLinkValue(link);
74     /////                    newValue[Constants.IndexPart] = Constants.Itself;
75     /////                    newValue[Constants.SourcePart] =
    ↳ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
    ↳ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
76     /////                    newValue[Constants.TargetPart] =
    ↳ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
    ↳ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
77     /////                    var matchDecision = matchedHandler(matchedLink, newValue);
78     /////                    if (Equals(matchDecision, Constants.Break))
79     /////                    {
80     /////                        return false;
81     /////                    }
82     /////                    if (!Equals(matchDecision, Constants.Skip))
83     /////                    {
84     /////                        transitions.Add(new Transition(matchedLink, newValue));
85     /////                    }
86     /////                    return true;
87     /////                };
88     /////                if (!Memory.Each(handler, restriction))
89     /////                {
90     /////                    return Constants.Break;
91     /////                }
92     /////            }
93     /////        }
94     /////    }
95     /////    else
96     /////    {
97     /////        Func<T, bool> handler = link =>
98     /////        {
99     /////            var matchedLink = Memory.GetLinkValue(link);
100     /////            var matchDecision = matchedHandler(matchedLink, matchedLink);
101     /////            return !Equals(matchDecision, Constants.Break);
102     /////        };
103     /////        if (!Memory.Each(handler, restriction))
104     /////        {
105     /////            return Constants.Break;
106     /////        }
107     /////    }
108     /////    if (substitution != null)
109     /////    {
110     /////        transitions = new List<Transition>();
111     /////        Func<T, bool> handler = link =>

```



```

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

```

```

176         //         return Constants.Break;
177         //         if (Equals(substitutedDecision, Constants.Continue))
178         //         {
179         //             // Actual update here
180         //             Memory.SetLinkValue(newValue);
181         //         }
182         //         if (Equals(substitutedDecision, Constants.Skip))
183         //         {
184         //             // Cancel the update. TODO: decide use separate Cancel
185         //             // constant or Skip is enough?
186         //         }
187         //     }
188     }
189 }
190 return _constants.Continue;
191 }
192
193 public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
194     ↳ matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
195     ↳ substitutionHandler)
196 {
197     var constants = _constants;
198     if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
199     {
200         return constants.Continue;
201     }
202     else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
203     ↳ Check if it is a correct condition
204     {
205         // Or it only applies to trigger without matchHandler.
206         throw new NotImplementedException();
207     }
208     else if (!substitution.IsNullOrEmpty()) // Creation
209     {
210         var before = Array.Empty<TLink>();
211         // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
212         ↳ (пройти мимо) или пустить (взять)?
213         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
214             ↳ constants.Break))
215         {
216             return constants.Break;
217         }
218         var after = (IList<TLink>)substitution.ToArray();
219         if (_equalityComparer.Equals(after[0], default))
220         {
221             var newLink = _links.Create();
222             after[0] = newLink;
223         }
224         if (substitution.Count == 1)
225         {
226             after = _links.GetLink(substitution[0]);
227         }
228         else if (substitution.Count == 3)
229         {
230             //Links.Create(after);
231         }
232         else
233         {
234             throw new NotSupportedException();
235         }
236         if (matchHandler != null)
237         {
238             return substitutionHandler(before, after);
239         }
240         return constants.Continue;
241     }
242     else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
243     {
244         if (patternOrCondition.Count == 1)
245         {
246             var linkToDelete = patternOrCondition[0];
247             var before = _links.GetLink(linkToDelete);
248             if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
249                 ↳ constants.Break))
250             {
251                 return constants.Break;
252             }
253         }
254     }

```

```

var after = Array.Empty<TLink>();
_links.Update(linkToDelete, constants.Null, constants.Null);
_links.Delete(linkToDelete);
if (matchHandler != null)
{
    return substitutionHandler(before, after);
}
return constants.Continue;
}
else
{
    throw new NotSupportedException();
}
}
else // Replace / Update
{
    if (patternOrCondition.Count == 1) //-V3125
    {
        var linkToUpdate = patternOrCondition[0];
        var before = _links.GetLink(linkToUpdate);
        if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
            ↪ constants.Break))
        {
            return constants.Break;
        }
        var after = (IList<TLink>)substitution.ToArray(); //-V3125
        if (_equalityComparer.Equals(after[0], default))
        {
            after[0] = linkToUpdate;
        }
        if (substitution.Count == 1)
        {
            if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
            {
                after = _links.GetLink(substitution[0]);
                _links.Update(linkToUpdate, constants.Null, constants.Null);
                _links.Delete(linkToUpdate);
            }
        }
        else if (substitution.Count == 3)
        {
            //Links.Update(after);
        }
        else
        {
            throw new NotSupportedException();
        }
        if (matchHandler != null)
        {
            return substitutionHandler(before, after);
        }
        return constants.Continue;
    }
    else
    {
        throw new NotSupportedException();
    }
}
}

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

```

```

323         changes.Add(change);
324         return @continue;
325     });
326     return changes;
327 }
328
329     private TLink AlwaysContinue(IList<TLink> linkToMatch) => _constants.Continue;
330 }
331 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets
8  {
9      public struct Doublet<T> : IEquatable<Doublet<T>>
10     {
11         private static readonly EqualityComparer<T> _equalityComparer =
12             EqualityComparer<T>.Default;
13
14         public T Source
15         {
16             [MethodImpl(MethodImplOptions.AggressiveInlining)]
17             get;
18             [MethodImpl(MethodImplOptions.AggressiveInlining)]
19             set;
20         }
21         public T Target
22         {
23             [MethodImpl(MethodImplOptions.AggressiveInlining)]
24             get;
25             [MethodImpl(MethodImplOptions.AggressiveInlining)]
26             set;
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public Doublet(T source, T target)
31         {
32             Source = source;
33             Target = target;
34         }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public override string ToString() => $"{Source}->{Target}";
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
41             && _equalityComparer.Equals(Target, other.Target);
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         public override bool Equals(object obj) => obj is Doublet<T> doublet ?
45             base.Equals(doublet) : false;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         public override int GetHashCode() => (Source, Target).GetHashCode();
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public static bool operator ==(Doublet<T> left, Doublet<T> right) => left.Equals(right);
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         public static bool operator !=(Doublet<T> left, Doublet<T> right) => !(left == right);
55     }
56 }

```

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

```

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

```

```

12 public class DoubletComparer<T> : IEqualityComparer<Doublet<T>>
13 {
14     public static readonly DoubletComparer<T> Default = new DoubletComparer<T>();
15
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     public bool Equals(Doublet<T> x, Doublet<T> y) => x.Equals(y);
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     public int GetHashCode(Doublet<T> obj) => obj.GetHashCode();
21 }
22 }

```

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

```

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

```

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

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Runtime.CompilerServices;
6 using Platform.Ranges;
7 using Platform.Collections.Arrays;
8 using Platform.Random;
9 using Platform.Setters;
10 using Platform.Converters;
11 using Platform.Numbers;
12 using Platform.Data.Exceptions;
13 using Platform.Data.Doublets.Decorators;
14
15 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
16
17 namespace Platform.Data.Doublets
18 {
19     public static class ILinksExtensions
20     {
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public static void RunRandomCreations<TLink>(this ILinks<TLink> links, ulong
23             ↪ amountOfCreations)
24         {
25             var random = RandomHelpers.Default;
26             var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
27             var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
28             for (var i = 0UL; i < amountOfCreations; i++)
29             {
30                 var linksAddressRange = new Range<ulong>(0,
31                     ↪ addressToUInt64Converter.Convert(links.Count()));
32                 var source =
33                     ↪ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
34                 var target =
35                     ↪ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
36                 links.GetOrCreate(source, target);
37             }
38         }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public static void RunRandomSearches<TLink>(this ILinks<TLink> links, ulong
42             ↪ amountOfSearches)
43         {
44             var random = RandomHelpers.Default;
45             var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
46             var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
47             for (var i = 0UL; i < amountOfSearches; i++)
48             {
49                 var linksAddressRange = new Range<ulong>(0,
50                     ↪ addressToUInt64Converter.Convert(links.Count()));
51                 var source =
52                     ↪ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
53                 var target =
54                     ↪ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
55                 links.SearchOrDefault(source, target);
56             }
57         }
58     }
59 }

```

```

48     }
49 }
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, ulong
↳ amountOfDeletions)
53 {
54     var random = RandomHelpers.Default;
55     var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
56     var uint64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
57     var linksCount = addressToUInt64Converter.Convert(links.Count());
58     var min = amountOfDeletions > linksCount ? OUL : linksCount - amountOfDeletions;
59     for (var i = OUL; i < amountOfDeletions; i++)
60     {
61         linksCount = addressToUInt64Converter.Convert(links.Count());
62         if (linksCount <= min)
63         {
64             break;
65         }
66         var linksAddressRange = new Range<ulong>(min, linksCount);
67         var link =
↳ uint64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
68         links.Delete(link);
69     }
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 public static void Delete<TLink>(this ILinks<TLink> links, TLink linkToDelete) =>
↳ links.Delete(new LinkAddress<TLink>(linkToDelete));
74
75 /// <remarks>
76 /// TODO: Возможно есть очень простой способ это сделать.
77 /// (Например просто удалить файл, или изменить его размер таким образом,
78 /// чтобы удалился весь контент)
79 /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
80 /// </remarks>
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 public static void DeleteAll<TLink>(this ILinks<TLink> links)
83 {
84     var equalityComparer = EqualityComparer<TLink>.Default;
85     var comparer = Comparer<TLink>.Default;
86     for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
↳ Arithmetic.Decrement(i))
87     {
88         links.Delete(i);
89         if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
90         {
91             i = links.Count();
92         }
93     }
94 }
95
96 [MethodImpl(MethodImplOptions.AggressiveInlining)]
97 public static TLink First<TLink>(this ILinks<TLink> links)
98 {
99     TLink firstLink = default;
100     var equalityComparer = EqualityComparer<TLink>.Default;
101     if (equalityComparer.Equals(links.Count(), default))
102     {
103         throw new InvalidOperationException("В хранилище нет связей.");
104     }
105     links.Each(links.Constants.Any, links.Constants.Any, link =>
106     {
107         firstLink = link[links.Constants.IndexPart];
108         return links.Constants.Break;
109     });
110     if (equalityComparer.Equals(firstLink, default))
111     {
112         throw new InvalidOperationException("В процессе поиска по хранилищу не было
↳ найдено связей.");
113     }
114     return firstLink;
115 }
116
117 #region Paths
118
119 /// <remarks>
120 /// TODO: Как так? Как то что ниже может быть корректно?
121 /// Скорее всего практически не применимо

```

```

122  /// Предполагалось, что можно было конвертировать формируемый в проходе через
123  ↪ SequenceWalker
124  /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
125  /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
126  /// </remarks>
127  [MethodImpl(MethodImplOptions.AggressiveInlining)]
128  public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
129  ↪ path)
130  {
131      var current = path[0];
132      //EnsureLinkExists(current, "path");
133      if (!links.Exists(current))
134      {
135          return false;
136      }
137      var equalityComparer = EqualityComparer<TLink>.Default;
138      var constants = links.Constants;
139      for (var i = 1; i < path.Length; i++)
140      {
141          var next = path[i];
142          var values = links.GetLink(current);
143          var source = values[constants.SourcePart];
144          var target = values[constants.TargetPart];
145          if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
146          ↪ next))
147          {
148              //throw new InvalidOperationException(string.Format("Невозможно выбрать
149              ↪ путь, так как и Source и Target совпадают с элементом пути {0}.", next));
150              return false;
151          }
152          if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
153          ↪ target))
154          {
155              //throw new InvalidOperationException(string.Format("Невозможно продолжить
156              ↪ путь через элемент пути {0}", next));
157              return false;
158          }
159          current = next;
160      }
161      return true;
162  }
163
164  /// <remarks>
165  /// Может потребовать дополнительного стека для PathElement's при использовании
166  ↪ SequenceWalker.
167  /// </remarks>
168  [MethodImpl(MethodImplOptions.AggressiveInlining)]
169  public static TLink GetByKeyes<TLink>(this ILinks<TLink> links, TLink root, params int[]
170  ↪ path)
171  {
172      links.EnsureLinkExists(root, "root");
173      var currentLink = root;
174      for (var i = 0; i < path.Length; i++)
175      {
176          currentLink = links.GetLink(currentLink)[path[i]];
177      }
178      return currentLink;
179  }
180
181  [MethodImpl(MethodImplOptions.AggressiveInlining)]
182  public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
183  ↪ links, TLink root, ulong size, ulong index)
184  {
185      var constants = links.Constants;
186      var source = constants.SourcePart;
187      var target = constants.TargetPart;
188      if (!Platform.Numbers.Math.IsPowerOfTwo(size))
189      {
190          throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
191          ↪ than powers of two are not supported.");
192      }
193      var path = new BitArray(BitConverter.GetBytes(index));
194      var length = Bit.GetLowestPosition(size);
195      links.EnsureLinkExists(root, "root");
196      var currentLink = root;
197      for (var i = length - 1; i >= 0; i--)
198      {
199          currentLink = links.GetLink(currentLink)[path[i] ? target : source];
200      }
201  }

```

```

190     }
191     return currentLink;
192 }
193
194 #endregion
195
196 /// <summary>
197 /// Возвращает индекс указанной связи.
198 /// </summary>
199 /// <param name="links">Хранилище связей.</param>
200 /// <param name="link">Связь представленная списком, состоящим из её адреса и
    → содержимого.</param>
201 /// <returns>Индекс начальной связи для указанной связи.</returns>
202 [MethodImpl(MethodImplOptions.AggressiveInlining)]
203 public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    → link[links.Constants.IndexPart];
204
205 /// <summary>
206 /// Возвращает индекс начальной (Source) связи для указанной связи.
207 /// </summary>
208 /// <param name="links">Хранилище связей.</param>
209 /// <param name="link">Индекс связи.</param>
210 /// <returns>Индекс начальной связи для указанной связи.</returns>
211 [MethodImpl(MethodImplOptions.AggressiveInlining)]
212 public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
    → links.GetLink(link)[links.Constants.SourcePart];
213
214 /// <summary>
215 /// Возвращает индекс начальной (Source) связи для указанной связи.
216 /// </summary>
217 /// <param name="links">Хранилище связей.</param>
218 /// <param name="link">Связь представленная списком, состоящим из её адреса и
    → содержимого.</param>
219 /// <returns>Индекс начальной связи для указанной связи.</returns>
220 [MethodImpl(MethodImplOptions.AggressiveInlining)]
221 public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    → link[links.Constants.SourcePart];
222
223 /// <summary>
224 /// Возвращает индекс конечной (Target) связи для указанной связи.
225 /// </summary>
226 /// <param name="links">Хранилище связей.</param>
227 /// <param name="link">Индекс связи.</param>
228 /// <returns>Индекс конечной связи для указанной связи.</returns>
229 [MethodImpl(MethodImplOptions.AggressiveInlining)]
230 public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
    → links.GetLink(link)[links.Constants.TargetPart];
231
232 /// <summary>
233 /// Возвращает индекс конечной (Target) связи для указанной связи.
234 /// </summary>
235 /// <param name="links">Хранилище связей.</param>
236 /// <param name="link">Связь представленная списком, состоящим из её адреса и
    → содержимого.</param>
237 /// <returns>Индекс конечной связи для указанной связи.</returns>
238 [MethodImpl(MethodImplOptions.AggressiveInlining)]
239 public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    → link[links.Constants.TargetPart];
240
241 /// <summary>
242 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    → (handler) для каждой подходящей связи.
243 /// </summary>
244 /// <param name="links">Хранилище связей.</param>
245 /// <param name="handler">Обработчик каждой подходящей связи.</param>
246 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
    → может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
    → Any - отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
247 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    → случае.</returns>
248 [MethodImpl(MethodImplOptions.AggressiveInlining)]
249 public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
    → handler, params TLink[] restrictions)
    => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
    → links.Constants.Continue);
250
251 /// <summary>
252

```



```

253 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
254   ↳ (handler) для каждой подходящей связи.
255 /// </summary>
256 /// <param name="links">Хранилище связей.</param>
257 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
258   ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
259   ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
260 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
261   ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
262   ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
263 /// <param name="handler">Обработчик каждой подходящей связи.</param>
264 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
265   ↳ случае.</returns>
266 [MethodImpl(MethodImplOptions.AggressiveInlining)]
267 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
268   ↳ Func<TLink, bool> handler)
269 {
270     var constants = links.Constants;
271     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
272       ↳ constants.Break, constants.Any, source, target);
273 }
274
275 /// <summary>
276 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
277   ↳ (handler) для каждой подходящей связи.
278 /// </summary>
279 /// <param name="links">Хранилище связей.</param>
280 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
281   ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
282   ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
283 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
284   ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
285   ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
286 /// <param name="handler">Обработчик каждой подходящей связи.</param>
287 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
288   ↳ случае.</returns>
289 [MethodImpl(MethodImplOptions.AggressiveInlining)]
290 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
291   ↳ Func<IList<TLink>, TLink> handler) => links.Each(handler, links.Constants.Any,
292   ↳ source, target);
293
294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
296   ↳ restrictions)
297 {
298     var arraySize = CheckedConverter<TLink,
299       ↳ long>.Default.Convert(links.Count(restrictions));
300     if (arraySize > 0)
301     {
302         var array = new IList<TLink>[arraySize];
303         var filler = new ArrayFiller<IList<TLink>, TLink>(array,
304           ↳ links.Constants.Continue);
305         links.Each(filler.AddAndReturnConstant, restrictions);
306         return array;
307     }
308     else
309     {
310         return Array.Empty<IList<TLink>>();
311     }
312 }
313
314 [MethodImpl(MethodImplOptions.AggressiveInlining)]
315 public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
316   ↳ restrictions)
317 {
318     var arraySize = CheckedConverter<TLink,
319       ↳ long>.Default.Convert(links.Count(restrictions));
320     if (arraySize > 0)
321     {
322         var array = new TLink[arraySize];
323         var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
324         links.Each(filler.AddFirstAndReturnConstant, restrictions);
325         return array;
326     }
327     else
328     {
329         return Array.Empty<TLink>();
330     }
331 }

```

```

309     }
310 }
311
312 /// <summary>
313 /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
314   ↳ в хранилище связей.
315 /// </summary>
316 /// <param name="links">Хранилище связей.</param>
317 /// <param name="source">Начало связи.</param>
318 /// <param name="target">Конец связи.</param>
319 /// <returns>Значение, определяющее существует ли связь.</returns>
320 [MethodImpl(MethodImplOptions.AggressiveInlining)]
321 public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
322   ↳ => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
323   ↳ default) > 0;
324
325 #region Ensure
326 // TODO: May be move to EnsureExtensions or make it both there and here
327
328 [MethodImpl(MethodImplOptions.AggressiveInlining)]
329 public static void EnsureLinkExists<TLink>(this ILinks<TLink> links, IList<TLink>
330   ↳ restrictions)
331 {
332     for (var i = 0; i < restrictions.Count; i++)
333     {
334         if (!links.Exists(restrictions[i]))
335         {
336             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
337               ↳ $"sequence[{i}]");
338         }
339     }
340 }
341
342 [MethodImpl(MethodImplOptions.AggressiveInlining)]
343 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
344   ↳ reference, string argumentName)
345 {
346     if (links.Constants.IsInternalReference(reference) && !links.Exists(reference))
347     {
348         throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
349     }
350 }
351
352 [MethodImpl(MethodImplOptions.AggressiveInlining)]
353 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
354   ↳ IList<TLink> restrictions, string argumentName)
355 {
356     for (int i = 0; i < restrictions.Count; i++)
357     {
358         links.EnsureInnerReferenceExists(restrictions[i], argumentName);
359     }
360 }
361
362 [MethodImpl(MethodImplOptions.AggressiveInlining)]
363 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
364   ↳ restrictions)
365 {
366     var equalityComparer = EqualityComparer<TLink>.Default;
367     var any = links.Constants.Any;
368     for (var i = 0; i < restrictions.Count; i++)
369     {
370         if (!equalityComparer.Equals(restrictions[i], any) &&
371           ↳ !links.Exists(restrictions[i]))
372         {
373             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
374               ↳ $"sequence[{i}]");
375         }
376     }
377 }
378
379 [MethodImpl(MethodImplOptions.AggressiveInlining)]
380 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
381   ↳ string argumentName)
382 {
383     var equalityComparer = EqualityComparer<TLink>.Default;
384     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
385     {
386

```

```

375         throw new ArgumentException<TLink>(link, argumentName);
376     }
377 }
378
379 [MethodImpl(MethodImplOptions.AggressiveInlining)]
380 public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
    ↳ link, string argumentName)
381 {
382     var equalityComparer = EqualityComparer<TLink>.Default;
383     if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
384     {
385         throw new ArgumentException<TLink>(link, argumentName);
386     }
387 }
388
389 /// <param name="links">Хранилище связей.</param>
390 [MethodImpl(MethodImplOptions.AggressiveInlining)]
391 public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
    ↳ TLink target)
392 {
393     if (links.Exists(source, target))
394     {
395         throw new LinkWithSameValueAlreadyExistsException();
396     }
397 }
398
399 /// <param name="links">Хранилище связей.</param>
400 [MethodImpl(MethodImplOptions.AggressiveInlining)]
401 public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
402 {
403     if (links.HasUsages(link))
404     {
405         throw new ArgumentException<TLink>(link);
406     }
407 }
408
409 /// <param name="links">Хранилище связей.</param>
410 [MethodImpl(MethodImplOptions.AggressiveInlining)]
411 public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ addresses) => links.EnsureCreated(links.Create, addresses);
412
413 /// <param name="links">Хранилище связей.</param>
414 [MethodImpl(MethodImplOptions.AggressiveInlining)]
415 public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
416
417 /// <param name="links">Хранилище связей.</param>
418 [MethodImpl(MethodImplOptions.AggressiveInlining)]
419 public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
    ↳ params TLink[] addresses)
420 {
421     var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
422     var uint64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
423     var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
    ↳ !links.Exists(x)));
424     if (nonExistentAddresses.Count > 0)
425     {
426         var max = nonExistentAddresses.Max();
427         max = uint64ToAddressConverter.Convert(System.Math.Min(addressToUInt64Converter.
    ↳ Convert(max),
    ↳ addressToUInt64Converter.Convert(links.Constants.InternalReferencesRange.Max
    ↳ imum)));
428         var createdLinks = new List<TLink>();
429         var equalityComparer = EqualityComparer<TLink>.Default;
430         TLink createdLink = creator();
431         while (!equalityComparer.Equals(createdLink, max))
432         {
433             createdLinks.Add(createdLink);
434         }
435         for (var i = 0; i < createdLinks.Count; i++)
436         {
437             if (!nonExistentAddresses.Contains(createdLinks[i]))
438             {
439                 links.Delete(createdLinks[i]);
440             }
441         }
442     }
443 }

```

```

444 #endregion
445
446
447 /// <param name="links">Хранилище связей.</param>
448 [MethodImpl(MethodImplOptions.AggressiveInlining)]
449 public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
450 {
451     var constants = links.Constants;
452     var values = links.GetLink(link);
453     TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link,
454         ↪ constants.Any));
455     var equalityComparer = EqualityComparer<TLink>.Default;
456     if (equalityComparer.Equals(values[constants.SourcePart], link))
457     {
458         usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
459     }
460     TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
461         ↪ link));
462     if (equalityComparer.Equals(values[constants.TargetPart], link))
463     {
464         usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
465     }
466     return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
467 }
468
469 /// <param name="links">Хранилище связей.</param>
470 [MethodImpl(MethodImplOptions.AggressiveInlining)]
471 public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
472     ↪ Comparer<TLink>.Default.Compare(links.CountUsages(link), default) > 0;
473
474 /// <param name="links">Хранилище связей.</param>
475 [MethodImpl(MethodImplOptions.AggressiveInlining)]
476 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
477     ↪ TLink target)
478 {
479     var constants = links.Constants;
480     var values = links.GetLink(link);
481     var equalityComparer = EqualityComparer<TLink>.Default;
482     return equalityComparer.Equals(values[constants.SourcePart], source) &&
483         ↪ equalityComparer.Equals(values[constants.TargetPart], target);
484 }
485
486 /// <summary>
487 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
488 /// </summary>
489 /// <param name="links">Хранилище связей.</param>
490 /// <param name="source">Индекс связи, которая является началом для искомой
491     ↪ связи.</param>
492 /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
493 /// <returns>Индекс искомой связи с указанными Source (началом) и Target
494     ↪ (концом).</returns>
495 [MethodImpl(MethodImplOptions.AggressiveInlining)]
496 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
497     ↪ target)
498 {
499     var constants = links.Constants;
500     var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break, default);
501     links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
502     return setter.Result;
503 }
504
505 /// <param name="links">Хранилище связей.</param>
506 [MethodImpl(MethodImplOptions.AggressiveInlining)]
507 public static TLink Create<TLink>(this ILinks<TLink> links) => links.Create(null);
508
509 /// <param name="links">Хранилище связей.</param>
510 [MethodImpl(MethodImplOptions.AggressiveInlining)]
511 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
512 {
513     var link = links.Create();
514     return links.Update(link, link, link);
515 }
516
517 /// <param name="links">Хранилище связей.</param>
518 [MethodImpl(MethodImplOptions.AggressiveInlining)]
519 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
520     ↪ target) => links.Update(links.Create(), source, target);

```

```

513 /// <summary>
514 /// Обновляет связь с указанными началом (Source) и концом (Target)
515 /// на связь с указанными началом (NewSource) и концом (NewTarget).
516 /// </summary>
517 /// <param name="links">Хранилище связей.</param>
518 /// <param name="link">Индекс обновляемой связи.</param>
519 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    ↳ выполняется обновление.</param>
520 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    ↳ выполняется обновление.</param>
521 /// <returns>Индекс обновлённой связи.</returns>
522 [MethodImpl(MethodImplOptions.AggressiveInlining)]
523 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));

524 /// <summary>
525 /// Обновляет связь с указанными началом (Source) и концом (Target)
526 /// на связь с указанными началом (NewSource) и концом (NewTarget).
527 /// </summary>
528 /// <param name="links">Хранилище связей.</param>
529 /// <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.. $\infty$  конкретный адрес другой
    ↳ связи.</param>
531 /// <returns>Индекс обновлённой связи.</returns>
532 [MethodImpl(MethodImplOptions.AggressiveInlining)]
533 public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
534 {
535     if (restrictions.Length == 2)
536     {
537         return links.MergeAndDelete(restrictions[0], restrictions[1]);
538     }
539     if (restrictions.Length == 4)
540     {
541         return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
    ↳ restrictions[2], restrictions[3]);
542     }
543     else
544     {
545         return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
546     }
547 }

548 [MethodImpl(MethodImplOptions.AggressiveInlining)]
549 public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
    ↳ links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
550 {
551     var equalityComparer = EqualityComparer<TLink>.Default;
552     var constants = links.Constants;
553     var restrictionsIndex = restrictions[constants.IndexPart];
554     var substitutionIndex = substitution[constants.IndexPart];
555     if (equalityComparer.Equals(substitutionIndex, default))
556     {
557         substitutionIndex = restrictionsIndex;
558     }
559     var source = substitution[constants.SourcePart];
560     var target = substitution[constants.TargetPart];
561     source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
562     target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
563     return new Link<TLink>(substitutionIndex, source, target);
564 }

565 /// <summary>
566 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
    ↳ с указанными Source (началом) и Target (концом).
567 /// </summary>
568 /// <param name="links">Хранилище связей.</param>
569 /// <param name="source">Индекс связи, которая является началом на создаваемой
    ↳ связи.</param>
570 /// <param name="target">Индекс связи, которая является концом для создаваемой
    ↳ связи.</param>
571 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
572 [MethodImpl(MethodImplOptions.AggressiveInlining)]
573 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target)
574 {
575

```

```

577     var link = links.SearchOrDefault(source, target);
578     if (EqualityComparer<TLink>.Default.Equals(link, default))
579     {
580         link = links.CreateAndUpdate(source, target);
581     }
582     return link;
583 }
584
585 /// <summary>
586 /// Обновляет связь с указанными началом (Source) и концом (Target)
587 /// на связь с указанными началом (NewSource) и концом (NewTarget).
588 /// </summary>
589 /// <param name="links">Хранилище связей.</param>
590 /// <param name="source">Индекс связи, которая является началом обновляемой
    → связи.</param>
591 /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
592 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    → выполняется обновление.</param>
593 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    → выполняется обновление.</param>
594 /// <returns>Индекс обновлённой связи.</returns>
595 [MethodImpl(MethodImplOptions.AggressiveInlining)]
596 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
    → TLink target, TLink newSource, TLink newTarget)
597 {
598     var equalityComparer = EqualityComparer<TLink>.Default;
599     var link = links.SearchOrDefault(source, target);
600     if (equalityComparer.Equals(link, default))
601     {
602         return links.CreateAndUpdate(newSource, newTarget);
603     }
604     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
    → target))
605     {
606         return link;
607     }
608     return links.Update(link, newSource, newTarget);
609 }
610
611 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
612 /// <param name="links">Хранилище связей.</param>
613 /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
614 /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
615 [MethodImpl(MethodImplOptions.AggressiveInlining)]
616 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
    → target)
617 {
618     var link = links.SearchOrDefault(source, target);
619     if (!EqualityComparer<TLink>.Default.Equals(link, default))
620     {
621         links.Delete(link);
622         return link;
623     }
624     return default;
625 }
626
627 /// <summary>Удаляет несколько связей.</summary>
628 /// <param name="links">Хранилище связей.</param>
629 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
630 [MethodImpl(MethodImplOptions.AggressiveInlining)]
631 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
632 {
633     for (int i = 0; i < deletedLinks.Count; i++)
634     {
635         links.Delete(deletedLinks[i]);
636     }
637 }
638
639 /// <remarks>Before execution of this method ensure that deleted link is detached (all
    → values - source and target are reset to null) or it might enter into infinite
    → recursion.</remarks>
640 [MethodImpl(MethodImplOptions.AggressiveInlining)]
641 public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
642 {
643     var anyConstant = links.Constants.Any;
644     var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
645     links.DeleteByQuery(usagesAsSourceQuery);
646     var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);

```

```

647     links.DeleteByQuery(usagesAsTargetQuery);
648 }
649
650 [MethodImpl(MethodImplOptions.AggressiveInlining)]
651 public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
652 {
653     var count = CheckedConverter<TLink, long>.Default.Convert(links.Count(query));
654     if (count > 0)
655     {
656         var queryResult = new TLink[count];
657         var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
658             ↪ links.Constants.Continue);
659         links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
660         for (var i = count - 1; i >= 0; i--)
661         {
662             links.Delete(queryResult[i]);
663         }
664     }
665
666     // TODO: Move to Platform.Data
667     [MethodImpl(MethodImplOptions.AggressiveInlining)]
668     public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
669     {
670         var nullConstant = links.Constants.Null;
671         var equalityComparer = EqualityComparer<TLink>.Default;
672         var link = links.GetLink(linkIndex);
673         for (int i = 1; i < link.Count; i++)
674         {
675             if (!equalityComparer.Equals(link[i], nullConstant))
676             {
677                 return false;
678             }
679         }
680         return true;
681     }
682
683     // TODO: Create a universal version of this method in Platform.Data (with using of for
684     ↪ loop)
685     [MethodImpl(MethodImplOptions.AggressiveInlining)]
686     public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
687     {
688         var nullConstant = links.Constants.Null;
689         var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
690         links.Update(updateRequest);
691     }
692
693     // TODO: Create a universal version of this method in Platform.Data (with using of for
694     ↪ loop)
695     [MethodImpl(MethodImplOptions.AggressiveInlining)]
696     public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
697     {
698         if (!links.AreValuesReset(linkIndex))
699         {
700             links.ResetValues(linkIndex);
701         }
702     }
703
704     /// <summary>
705     /// Merging two usages graphs, all children of old link moved to be children of new link
706     ↪ or deleted.
707     /// </summary>
708     [MethodImpl(MethodImplOptions.AggressiveInlining)]
709     public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
710     ↪ TLink newLinkIndex)
711     {
712         var addressToInt64Converter = CheckedConverter<TLink, long>.Default;
713         var equalityComparer = EqualityComparer<TLink>.Default;
714         if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
715         {
716             var constants = links.Constants;
717             var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
718                 ↪ constants.Any);
719             var usagesAsSourceCount =
720                 ↪ addressToInt64Converter.Convert(links.Count(usagesAsSourceQuery));
721             var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
722                 ↪ oldLinkIndex);

```

```

716     var usagesAsTargetCount =
717         ↪ addressToInt64Converter.Convert(links.Count(usagesAsTargetQuery));
718     var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
719         ↪ usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
720     if (!isStandalonePoint)
721     {
722         var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
723         if (totalUsages > 0)
724         {
725             var usages = ArrayPool.Allocate<TLink>(totalUsages);
726             var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
727                 ↪ links.Constants.Continue);
728             var i = 0L;
729             if (usagesAsSourceCount > 0)
730             {
731                 links.Each(usagesFiller.AddFirstAndReturnConstant,
732                     ↪ usagesAsSourceQuery);
733                 for (; i < usagesAsSourceCount; i++)
734                 {
735                     var usage = usages[i];
736                     if (!equalityComparer.Equals(usage, oldLinkIndex))
737                     {
738                         links.Update(usage, newLinkIndex, links.GetTarget(usage));
739                     }
740                 }
741             }
742             if (usagesAsTargetCount > 0)
743             {
744                 links.Each(usagesFiller.AddFirstAndReturnConstant,
745                     ↪ usagesAsTargetQuery);
746                 for (; i < usages.Length; i++)
747                 {
748                     var usage = usages[i];
749                     if (!equalityComparer.Equals(usage, oldLinkIndex))
750                     {
751                         links.Update(usage, links.GetSource(usage), newLinkIndex);
752                     }
753                 }
754             }
755             ArrayPool.Free(usages);
756         }
757     }
758     return newLinkIndex;
759 }
760
761 /// <summary>
762 /// Replace one link with another (replaced link is deleted, children are updated or
763 /// ↪ deleted).
764 /// </summary>
765 [MethodImpl(MethodImplOptions.AggressiveInlining)]
766 public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
767     ↪ TLink newLinkIndex)
768 {
769     var equalityComparer = EqualityComparer<TLink>.Default;
770     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
771     {
772         links.MergeUsages(oldLinkIndex, newLinkIndex);
773         links.Delete(oldLinkIndex);
774     }
775     return newLinkIndex;
776 }
777
778 [MethodImpl(MethodImplOptions.AggressiveInlining)]
779 public static ILinks<TLink>
780     ↪ DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
781 {
782     links = new LinksCascadeUsagesResolver<TLink>(links);
783     links = new NonNullContentsLinkDeletionResolver<TLink>(links);
784     links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
785     return links;
786 }
787 }

```

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

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2

```



```

3 namespace Platform.Data.Doublets
4 {
5     public interface ISynchronizedLinks<TLink> : ISynchronizedLinks<TLink, ILinks<TLink>,
        ↳ LinksConstants<TLink>>, ILinks<TLink>
6     {
7     }
8 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Incrementers;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Incrementers
8 {
9     public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
10    {
11        private static readonly EqualityComparer<TLink> _equalityComparer =
12            ↳ EqualityComparer<TLink>.Default;
13
14        private readonly TLink _frequencyMarker;
15        private readonly TLink _unaryOne;
16        private readonly IIncrementer<TLink> _unaryNumberIncrementer;
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
20            ↳ IIncrementer<TLink> unaryNumberIncrementer)
21            : base(links)
22        {
23            _frequencyMarker = frequencyMarker;
24            _unaryOne = unaryOne;
25            _unaryNumberIncrementer = unaryNumberIncrementer;
26        }
27
28        [MethodImpl(MethodImplOptions.AggressiveInlining)]
29        public TLink Increment(TLink frequency)
30        {
31            var links = _links;
32            if (_equalityComparer.Equals(frequency, default))
33            {
34                return links.GetOrCreate(_unaryOne, _frequencyMarker);
35            }
36            var incrementedSource =
37                ↳ _unaryNumberIncrementer.Increment(links.GetSource(frequency));
38            return links.GetOrCreate(incrementedSource, _frequencyMarker);
39        }
40    }
41 }

```

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

```

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

```

```

27         var target = links.GetTarget(unaryNumber);
28         if (_equalityComparer.Equals(source, target))
29         {
30             return links.GetOrCreate(unaryNumber, _unaryOne);
31         }
32         else
33         {
34             return links.GetOrCreate(source, Increment(target));
35         }
36     }
37 }
38 }

```

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

```

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

```

```

63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 private static void SetValues(ref Link<TLink> other, out TLink index, out TLink source,
65     ↪ out TLink target)
66 {
67     index = other.Index;
68     source = other.Source;
69     target = other.Target;
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 private static void SetValues(IList<TLink> values, out TLink index, out TLink source,
74     ↪ out TLink target)
75 {
76     switch (values.Count)
77     {
78         case 3:
79             index = values[0];
80             source = values[1];
81             target = values[2];
82             break;
83         case 2:
84             index = values[0];
85             source = values[1];
86             target = default;
87             break;
88         case 1:
89             index = values[0];
90             source = default;
91             target = default;
92             break;
93         default:
94             index = default;
95             source = default;
96             target = default;
97             break;
98     }
99 }
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 public override int GetHashCode() => (Index, Source, Target).GetHashCode();
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
105     && _equalityComparer.Equals(Source, _constants.Null)
106     && _equalityComparer.Equals(Target, _constants.Null);
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 public override bool Equals(object other) => other is Link<TLink> &&
110     ↪ Equals((Link<TLink>)other);
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
114     && _equalityComparer.Equals(Source, other.Source)
115     && _equalityComparer.Equals(Target, other.Target);
116
117 [MethodImpl(MethodImplOptions.AggressiveInlining)]
118 public static string ToString(TLink index, TLink source, TLink target) => $"{index}:
119     ↪ {source}->{target}";
120
121 [MethodImpl(MethodImplOptions.AggressiveInlining)]
122 public static string ToString(TLink source, TLink target) => $"{source}->{target}";
123
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 public static implicit operator TLink[] (Link<TLink> link) => link.ToArray();
126
127 [MethodImpl(MethodImplOptions.AggressiveInlining)]
128 public static implicit operator Link<TLink> (TLink[] linkArray) => new
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 public int Count
137 {
138     [MethodImpl(MethodImplOptions.AggressiveInlining)]
139     get => Length;
140 }

```

```

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

```

```

213         if (_equalityComparer.Equals(Source, item))
214         {
215             return _constants.SourcePart;
216         }
217         if (_equalityComparer.Equals(Target, item))
218         {
219             return _constants.TargetPart;
220         }
221         return -1;
222     }
223
224     [MethodImpl(MethodImplOptions.AggressiveInlining)]
225     public void Insert(int index, TLink item) => throw new NotSupportedException();
226
227     [MethodImpl(MethodImplOptions.AggressiveInlining)]
228     public void RemoveAt(int index) => throw new NotSupportedException();
229
230     [MethodImpl(MethodImplOptions.AggressiveInlining)]
231     public static bool operator ==(Link<TLink> left, Link<TLink> right) =>
232         ↪ left.Equals(right);
233
234     [MethodImpl(MethodImplOptions.AggressiveInlining)]
235     public static bool operator !=(Link<TLink> left, Link<TLink> right) => !(left == right);
236
237     #endregion
238 }

```

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

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets
6  {
7      public static class LinkExtensions
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public static bool IsFullPoint<TLink>(this Link<TLink> link) =>
11             ↪ Point<TLink>.IsFullPoint(link);
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
15             ↪ Point<TLink>.IsPartialPoint(link);
16     }
17 }

```

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

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets
6  {
7      public abstract class LinksOperatorBase<TLink>
8      {
9          protected readonly ILinks<TLink> _links;
10
11          public ILinks<TLink> Links
12          {
13              [MethodImpl(MethodImplOptions.AggressiveInlining)]
14              get => _links;
15          }
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected LinksOperatorBase(ILinks<TLink> links) => _links = links;
19      }
20 }

```

1.25 ./csharp/Platform.Data.Doublets/Memory/ILinksListMethods.cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory
6  {
7      public interface ILinksListMethods<TLink>
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

10         void Detach(TLink freeLink);
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         void AttachAsFirst(TLink link);
14     }
15 }

```

1.26 ./csharp/Platform.Data.Doublets/Memory/ILinksTreeMethods.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Memory
8  {
9      public interface ILinksTreeMethods<TLink>
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         TLink CountUsages(TLink root);
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         TLink Search(TLink source, TLink target);
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         TLink EachUsage(TLink root, Func<IList<TLink>, TLink> handler);
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         void Detach(ref TLink root, TLink linkIndex);
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         void Attach(ref TLink root, TLink linkIndex);
25     }
26 }

```

1.27 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/LinksSizeBalancedTreeMethodsBase.cs

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7  using static System.Runtime.CompilerServices.Unsafe;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Memory.Split.Generic
12 {
13     public unsafe abstract class LinksSizeBalancedTreeMethodsBase<TLink> :
14         ↳ SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
15     {
16         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
17             ↳ UncheckedConverter<TLink, long>.Default;
18
19         protected readonly TLink Break;
20         protected readonly TLink Continue;
21         protected readonly byte* LinksDataParts;
22         protected readonly byte* LinksIndexParts;
23         protected readonly byte* Header;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte*
27             ↳ linksDataParts, byte* linksIndexParts, byte* header)
28         {
29             LinksDataParts = linksDataParts;
30             LinksIndexParts = linksIndexParts;
31             Header = header;
32             Break = constants.Break;
33             Continue = constants.Continue;
34         }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected abstract TLink GetTreeRoot(TLink link);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected abstract TLink GetBasePartValue(TLink link);
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

43     protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
44     ↪ AsRef<RawLinkDataPart<TLink>>(LinksDataParts + RawLinkDataPart<TLink>.SizeInBytes *
45     ↪ _addressToInt64Converter.Convert(link));
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
49     ↪ ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
50     ↪ RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link));
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
54     ↪ LessThan(GetKeyPartValue(first), GetKeyPartValue(second));
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
58     ↪ GreaterThan(GetKeyPartValue(first), GetKeyPartValue(second));
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
62     {
63         ref var link = ref GetLinkDataPartReference(linkIndex);
64         return new Link<TLink>(linkIndex, link.Source, link.Target);
65     }
66
67     public TLink this[TLink link, TLink index]
68     {
69         [MethodImpl(MethodImplOptions.AggressiveInlining)]
70         get
71         {
72             var root = GetTreeRoot(link);
73             if (GreaterOrEqualThan(index, GetSize(root)))
74             {
75                 return Zero;
76             }
77             while (!EqualToZero(root))
78             {
79                 var left = GetLeftOrDefault(root);
80                 var leftSize = GetSizeOrZero(left);
81                 if (LessThan(index, leftSize))
82                 {
83                     root = left;
84                     continue;
85                 }
86                 if (AreEqual(index, leftSize))
87                 {
88                     return root;
89                 }
90                 root = GetRightOrDefault(root);
91                 index = Subtract(index, Increment(leftSize));
92             }
93             return Zero; // TODO: Impossible situation exception (only if tree structure
94             ↪ broken)
95         }
96     }
97
98     /// <summary>
99     /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
100     ↪ (концом).
101     /// </summary>
102     /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
103     /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
104     /// <returns>Индекс искомой связи.</returns>
105     [MethodImpl(MethodImplOptions.AggressiveInlining)]
106     public abstract TLink Search(TLink source, TLink target);
107
108     [MethodImpl(MethodImplOptions.AggressiveInlining)]
109     protected TLink SearchCore(TLink root, TLink key)
110     {
111         while (!EqualToZero(root))
112         {
113             var rootKey = GetKeyPartValue(root);
114             if (LessThan(key, rootKey)) // node.Key < root.Key
115             {
116                 root = GetLeftOrDefault(root);
117             }
118             else if (GreaterThan(key, rootKey)) // node.Key > root.Key
119             {
120                 root = GetRightOrDefault(root);
121             }
122         }
123     }

```

```

113     }
114     else // node.Key == root.Key
115     {
116         return root;
117     }
118 }
119 return Zero;
120 }
121
122 // TODO: Return indices range instead of references count
123 [MethodImpl(MethodImplOptions.AggressiveInlining)]
124 public TLink CountUsages(TLink link) => GetSizeOrZero(GetTreeRoot(link));
125
126 [MethodImpl(MethodImplOptions.AggressiveInlining)]
127 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
    ↳ EachUsageCore(@base, GetTreeRoot(@base), handler);
128
129 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
    ↳ low-level MSIL stack.
130 [MethodImpl(MethodImplOptions.AggressiveInlining)]
131 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
132 {
133     var @continue = Continue;
134     if (EqualToZero(link))
135     {
136         return @continue;
137     }
138     var @break = Break;
139     if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
140     {
141         return @break;
142     }
143     if (AreEqual(handler(GetLinkValues(link)), @break))
144     {
145         return @break;
146     }
147     if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
148     {
149         return @break;
150     }
151     return @continue;
152 }
153
154 [MethodImpl(MethodImplOptions.AggressiveInlining)]
155 protected override void PrintNodeValue(TLink node, StringBuilder sb)
156 {
157     ref var link = ref GetLinkDataPartReference(node);
158     sb.Append(' ');
159     sb.Append(link.Source);
160     sb.Append('-');
161     sb.Append('>');
162     sb.Append(link.Target);
163 }
164 }
165 }

```

1.28 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/LinksSourcesSizeBalancedTreeMethods.cs

```

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

```



```

18 [MethodImpl(MethodImplOptions.AggressiveInlining)]
19 protected override TLink GetLeft(TLink node) =>
    ↳ GetLinkIndexPartReference(node).LeftAsSource;
20
21 [MethodImpl(MethodImplOptions.AggressiveInlining)]
22 protected override TLink GetRight(TLink node) =>
    ↳ GetLinkIndexPartReference(node).RightAsSource;
23
24 [MethodImpl(MethodImplOptions.AggressiveInlining)]
25 protected override void SetLeft(TLink node, TLink left) =>
    ↳ GetLinkIndexPartReference(node).LeftAsSource = left;
26
27 [MethodImpl(MethodImplOptions.AggressiveInlining)]
28 protected override void SetRight(TLink node, TLink right) =>
    ↳ GetLinkIndexPartReference(node).RightAsSource = right;
29
30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 protected override TLink GetSize(TLink node) =>
    ↳ GetLinkIndexPartReference(node).SizeAsSource;
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 protected override void SetSize(TLink node, TLink size) =>
    ↳ GetLinkIndexPartReference(node).SizeAsSource = size;
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 protected override TLink GetTreeRoot(TLink link) =>
    ↳ GetLinkIndexPartReference(link).RootAsSource;
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 protected override TLink GetBasePartValue(TLink link) =>
    ↳ GetLinkDataPartReference(link).Source;
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected override TLink GetKeyPartValue(TLink link) =>
    ↳ GetLinkDataPartReference(link).Target;
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 protected override void ClearNode(TLink node)
47 {
48     ref var link = ref GetLinkIndexPartReference(node);
49     link.LeftAsSource = Zero;
50     link.RightAsSource = Zero;
51     link.SizeAsSource = Zero;
52 }
53
54 public override TLink Search(TLink source, TLink target) =>
    ↳ SearchCore(GetTreeRoot(source), target);
55 }
56 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Memory.Split.Generic
6 {
7     public unsafe class LinksTargetsSizeBalancedTreeMethods<TLink> :
8         ↳ LinksSizeBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte*
12             ↳ linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
13             ↳ linksDataParts, linksIndexParts, header) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected override ref TLink GetLeftReference(TLink node) => ref
17             ↳ GetLinkIndexPartReference(node).LeftAsTarget;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected override ref TLink GetRightReference(TLink node) => ref
21             ↳ GetLinkIndexPartReference(node).RightAsTarget;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override TLink GetLeft(TLink node) =>
25             ↳ GetLinkIndexPartReference(node).LeftAsTarget;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override TLink GetRight(TLink node) =>
29             ↳ GetLinkIndexPartReference(node).RightAsTarget;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetLeft(TLink node, TLink left) =>
33             ↳ GetLinkIndexPartReference(node).LeftAsTarget = left;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override void SetRight(TLink node, TLink right) =>
37             ↳ GetLinkIndexPartReference(node).RightAsTarget = right;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(TLink node, TLink size) =>
41             ↳ GetLinkIndexPartReference(node).SizeAsSource = size;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override TLink GetTreeRoot(TLink link) =>
45             ↳ GetLinkIndexPartReference(link).RootAsSource;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override TLink GetBasePartValue(TLink link) =>
49             ↳ GetLinkDataPartReference(link).Source;
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override TLink GetKeyPartValue(TLink link) =>
53             ↳ GetLinkDataPartReference(link).Target;
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override void ClearNode(TLink node)
57         {
58             ref var link = ref GetLinkIndexPartReference(node);
59             link.LeftAsSource = Zero;
60             link.RightAsSource = Zero;
61             link.SizeAsSource = Zero;
62         }
63
64         public override TLink Search(TLink source, TLink target) =>
65             ↳ SearchCore(GetTreeRoot(source), target);
66     }
67 }

```

```

22     protected override TLink GetRight(TLink node) =>
23         ↪ GetLinkIndexPartReference(node).RightAsTarget;
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     protected override void SetLeft(TLink node, TLink left) =>
27         ↪ GetLinkIndexPartReference(node).LeftAsTarget = left;
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     protected override void SetRight(TLink node, TLink right) =>
31         ↪ GetLinkIndexPartReference(node).RightAsTarget = right;
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override TLink GetSize(TLink node) =>
35         ↪ GetLinkIndexPartReference(node).SizeAsTarget;
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override void SetSize(TLink node, TLink size) =>
39         ↪ GetLinkIndexPartReference(node).SizeAsTarget = size;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override TLink GetTreeRoot(TLink link) =>
43         ↪ GetLinkIndexPartReference(link).RootAsTarget;
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override TLink GetBasePartValue(TLink link) =>
47         ↪ GetLinkDataPartReference(link).Target;
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override TLink GetKeyPartValue(TLink link) =>
51         ↪ GetLinkDataPartReference(link).Source;
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override void ClearNode(TLink node)
55     {
56         ref var link = ref GetLinkIndexPartReference(node);
57         link.LeftAsTarget = Zero;
58         link.RightAsTarget = Zero;
59         link.SizeAsTarget = Zero;
60     }
61
62     public override TLink Search(TLink source, TLink target) =>
63         ↪ SearchCore(GetTreeRoot(target), source);
64 }

```

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

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using static System.Runtime.CompilerServices.Unsafe;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Memory.Split.Generic
10 {
11     public unsafe class SplitMemoryLinks<TLink> : SplitMemoryLinksBase<TLink>
12     {
13         private readonly Func<ILinksTreeMethods<TLink>> _createSourceTreeMethods;
14         private readonly Func<ILinksTreeMethods<TLink>> _createTargetTreeMethods;
15         private byte* _header;
16         private byte* _linksDataParts;
17         private byte* _linksIndexParts;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
21             ↪ indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
25             ↪ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
26             ↪ memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
30             ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
31             ↪ base(dataMemory, indexMemory, memoryReservationStep, constants)
32         {

```

```

28         _createSourceTreeMethods = () => new
        ↪ LinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
        ↪ _linksIndexParts, _header);
29         _createTargetTreeMethods = () => new
        ↪ LinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
        ↪ _linksIndexParts, _header);
30         Init(dataMemory, indexMemory, memoryReservationStep);
31     }
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override void SetPointers(IResizableDirectMemory dataMemory,
        ↪ IResizableDirectMemory indexMemory)
35     {
36         _linksDataParts = (byte*)dataMemory.Pointer;
37         _linksIndexParts = (byte*)indexMemory.Pointer;
38         _header = _linksIndexParts;
39         SourcesTreeMethods = _createSourceTreeMethods();
40         TargetsTreeMethods = _createTargetTreeMethods();
41         UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_linksDataParts, _header);
42     }
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override void ResetPointers()
46     {
47         base.ResetPointers();
48         _linksDataParts = null;
49         _linksIndexParts = null;
50         _header = null;
51     }
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override ref LinksHeaderIndexPart<TLink> GetHeaderReference() => ref
        ↪ AsRef<LinksHeaderIndexPart<TLink>>(_header);
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
        ↪ => ref AsRef<RawLinkDataPart<TLink>>(_linksDataParts + LinkDataPartSizeInBytes *
        ↪ ConvertToInt64(linkIndex));
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
        ↪ linkIndex) => ref AsRef<RawLinkIndexPart<TLink>>(_linksIndexParts +
        ↪ LinkIndexPartSizeInBytes * ConvertToInt64(linkIndex));
61 }
62 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5  using Platform.Singletons;
6  using Platform.Converters;
7  using Platform.Numbers;
8  using Platform.Memory;
9  using Platform.Data.Exceptions;
10
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Data.Doublets.Memory.Split.Generic
14 {
15     public abstract class SplitMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16     {
17         private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;
18         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
19         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
        ↪ UncheckedConverter<TLink, long>.Default;
20         private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
        ↪ UncheckedConverter<long, TLink>.Default;
21
22         private static readonly TLink _zero = default;
23         private static readonly TLink _one = Arithmetic.Increment(_zero);
24
25         /// <summary>Возвращает размер одной связи в байтах.</summary>
26         /// <remarks>
27         /// Используется только во вне класса, не рекомендуется использовать внутри.
28         /// Так как во вне не обязательно будет доступен unsafe C#.
29         /// </remarks>
30         public static readonly long LinkDataPartSizeInBytes = RawLinkDataPart<TLink>.SizeInBytes;

```

```

31
32 public static readonly long LinkIndexPartSizeInBytes =
    ↳ RawLinkIndexPart<TLink>.SizeInBytes;
33
34 public static readonly long LinkHeaderSizeInBytes =
    ↳ LinksHeaderIndexPart<TLink>.SizeInBytes;
35
36 public static readonly long DefaultLinksSizeStep = 1 * 1024 * 1024;
37
38 protected readonly IResizableDirectMemory _dataMemory;
39 protected readonly IResizableDirectMemory _indexMemory;
40 protected readonly long _dataMemoryReservationStepInBytes;
41 protected readonly long _indexMemoryReservationStepInBytes;
42
43 protected ILinksTreeMethods<TLink> TargetsTreeMethods;
44 protected ILinksTreeMethods<TLink> SourcesTreeMethods;
45 // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
    ↳ нужно использовать не список а дерево, так как так можно быстрее проверить на
    ↳ наличие связи внутри
46 protected ILinksListMethods<TLink> UnusedLinksListMethods;
47
48 /// <summary>
49 /// Возвращает общее число связей находящихся в хранилище.
50 /// </summary>
51 protected virtual TLink Total
52 {
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     get
55     {
56         ref var header = ref GetHeaderReference();
57         return Subtract(header.AllocatedLinks, header.FreeLinks);
58     }
59 }
60
61 public virtual LinksConstants<TLink> Constants
62 {
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     get;
65 }
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants)
69 {
70     _dataMemory = dataMemory;
71     _indexMemory = indexMemory;
72     _dataMemoryReservationStepInBytes = memoryReservationStep * LinkDataPartSizeInBytes;
73     _indexMemoryReservationStepInBytes = memoryReservationStep *
    ↳ LinkIndexPartSizeInBytes;
74     Constants = constants;
75 }
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
    ↳ memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 protected virtual void Init(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory, long memoryReservationStep)
82 {
83     if (dataMemory.ReservedCapacity < memoryReservationStep)
84     {
85         dataMemory.ReservedCapacity = memoryReservationStep;
86     }
87     if (indexMemory.ReservedCapacity < memoryReservationStep)
88     {
89         indexMemory.ReservedCapacity = memoryReservationStep;
90     }
91     SetPointers(dataMemory, indexMemory);
92     ref var header = ref GetHeaderReference();
93     // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
94     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
95     dataMemory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) *
    ↳ LinkDataPartSizeInBytes + LinkDataPartSizeInBytes; // First link is read only
    ↳ zero link.
96     indexMemory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) *
    ↳ LinkIndexPartSizeInBytes + LinkHeaderSizeInBytes;
97     // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
98     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity

```

```

99         header.ReservedLinks = ConvertToAddress((dataMemory.ReservedCapacity -
100             ↪ LinkDataPartSizeInBytes) / LinkDataPartSizeInBytes);
101     }
102     [MethodImpl(MethodImplOptions.AggressiveInlining)]
103     public virtual TLink Count(IList<TLink> restrictions)
104     {
105         // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
106         if (restrictions.Count == 0)
107         {
108             return Total;
109         }
110         var constants = Constants;
111         var any = constants.Any;
112         var index = restrictions[constants.IndexPart];
113         if (restrictions.Count == 1)
114         {
115             if (AreEqual(index, any))
116             {
117                 return Total;
118             }
119             return Exists(index) ? GetOne() : GetZero();
120         }
121         if (restrictions.Count == 2)
122         {
123             var value = restrictions[1];
124             if (AreEqual(index, any))
125             {
126                 if (AreEqual(value, any))
127                 {
128                     return Total; // Any - как отсутствие ограничения
129                 }
130                 return Add(SourcesTreeMethods.CountUsages(value),
131                     ↪ TargetsTreeMethods.CountUsages(value));
132             }
133             else
134             {
135                 if (!Exists(index))
136                 {
137                     return GetZero();
138                 }
139                 if (AreEqual(value, any))
140                 {
141                     return GetOne();
142                 }
143                 ref var storedLinkValue = ref GetLinkDataPartReference(index);
144                 if (AreEqual(storedLinkValue.Source, value) ||
145                     ↪ AreEqual(storedLinkValue.Target, value))
146                 {
147                     return GetOne();
148                 }
149                 return GetZero();
150             }
151         }
152         if (restrictions.Count == 3)
153         {
154             var source = restrictions[constants.SourcePart];
155             var target = restrictions[constants.TargetPart];
156             if (AreEqual(index, any))
157             {
158                 if (AreEqual(source, any) && AreEqual(target, any))
159                 {
160                     return Total;
161                 }
162                 else if (AreEqual(source, any))
163                 {
164                     return TargetsTreeMethods.CountUsages(target);
165                 }
166                 else if (AreEqual(target, any))
167                 {
168                     return SourcesTreeMethods.CountUsages(source);
169                 }
170                 else //if(source != Any && target != Any)
171                 {
172                     // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
173                     var link = SourcesTreeMethods.Search(source, target);
174                     return AreEqual(link, constants.Null) ? GetZero() : GetOne();
175                 }
176             }
177         }
178     }

```

```

174     }
175     else
176     {
177         if (!Exists(index))
178         {
179             return GetZero();
180         }
181         if (AreEqual(source, any) && AreEqual(target, any))
182         {
183             return GetOne();
184         }
185         ref var storedLinkValue = ref GetLinkDataPartReference(index);
186         if (!AreEqual(source, any) && !AreEqual(target, any))
187         {
188             if (AreEqual(storedLinkValue.Source, source) &&
189                 ↪ AreEqual(storedLinkValue.Target, target))
190             {
191                 return GetOne();
192             }
193             return GetZero();
194         }
195         var value = default(TLink);
196         if (AreEqual(source, any))
197         {
198             value = target;
199         }
200         if (AreEqual(target, any))
201         {
202             value = source;
203         }
204         if (AreEqual(storedLinkValue.Source, value) ||
205             ↪ AreEqual(storedLinkValue.Target, value))
206         {
207             return GetOne();
208         }
209         return GetZero();
210     }
211 }
212
213 [MethodImpl(MethodImplOptions.AggressiveInlining)]
214 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
215 {
216     var constants = Constants;
217     var @break = constants.Break;
218     if (restrictions.Count == 0)
219     {
220         for (var link = GetOne(); LessOrEqualThan(link,
221             ↪ GetHeaderReference().AllocatedLinks); link = Increment(link))
222         {
223             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
224             {
225                 return @break;
226             }
227         }
228         return @break;
229     }
230     var @continue = constants.Continue;
231     var any = constants.Any;
232     var index = restrictions[constants.IndexPart];
233     if (restrictions.Count == 1)
234     {
235         if (AreEqual(index, any))
236         {
237             return Each(handler, Array.Empty<TLink>());
238         }
239         if (!Exists(index))
240         {
241             return @continue;
242         }
243         return handler(GetLinkStruct(index));
244     }
245     if (restrictions.Count == 2)
246     {
247         var value = restrictions[1];
248         if (AreEqual(index, any))

```

```

248 {
249     if (AreEqual(value, any))
250     {
251         return Each(handler, Array.Empty<TLink>());
252     }
253     if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
254     {
255         return @break;
256     }
257     return Each(handler, new Link<TLink>(index, any, value));
258 }
259 else
260 {
261     if (!Exists(index))
262     {
263         return @continue;
264     }
265     if (AreEqual(value, any))
266     {
267         return handler(GetLinkStruct(index));
268     }
269     ref var storedLinkValue = ref GetLinkDataPartReference(index);
270     if (AreEqual(storedLinkValue.Source, value) ||
271         AreEqual(storedLinkValue.Target, value))
272     {
273         return handler(GetLinkStruct(index));
274     }
275     return @continue;
276 }
277 }
278 if (restrictions.Count == 3)
279 {
280     var source = restrictions[constants.SourcePart];
281     var target = restrictions[constants.TargetPart];
282     if (AreEqual(index, any))
283     {
284         if (AreEqual(source, any) && AreEqual(target, any))
285         {
286             return Each(handler, Array.Empty<TLink>());
287         }
288         else if (AreEqual(source, any))
289         {
290             return TargetsTreeMethods.EachUsage(target, handler);
291         }
292         else if (AreEqual(target, any))
293         {
294             return SourcesTreeMethods.EachUsage(source, handler);
295         }
296         else //if(source != Any && target != Any)
297         {
298             var link = SourcesTreeMethods.Search(source, target);
299             return AreEqual(link, constants.Null) ? @continue :
300                 ↪ handler(GetLinkStruct(link));
301         }
302     }
303     else
304     {
305         if (!Exists(index))
306         {
307             return @continue;
308         }
309         if (AreEqual(source, any) && AreEqual(target, any))
310         {
311             return handler(GetLinkStruct(index));
312         }
313         ref var storedLinkValue = ref GetLinkDataPartReference(index);
314         if (!AreEqual(source, any) && !AreEqual(target, any))
315         {
316             if (AreEqual(storedLinkValue.Source, source) &&
317                 AreEqual(storedLinkValue.Target, target))
318             {
319                 return handler(GetLinkStruct(index));
320             }
321             return @continue;
322         }
323         var value = default(TLink);
324         if (AreEqual(source, any))
325         {

```

```

325         value = target;
326     }
327     if (AreEqual(target, any))
328     {
329         value = source;
330     }
331     if (AreEqual(storedLinkValue.Source, value) ||
332         AreEqual(storedLinkValue.Target, value))
333     {
334         return handler(GetLinkStruct(index));
335     }
336     return @continue;
337 }
338 }
339 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
340 }
341
342 /// <remarks>
343 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↳ в другом месте (но не в менеджере памяти, а в логике Links)
344 /// </remarks>
345 [MethodImpl(MethodImplOptions.AggressiveInlining)]
346 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
347 {
348     var constants = Constants;
349     var @null = constants.Null;
350     var linkIndex = restrictions[constants.IndexPart];
351     ref var link = ref GetLinkDataPartReference(linkIndex);
352     ref var header = ref GetHeaderReference();
353     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↳ предварительно заполнено нулями
354     if (!AreEqual(link.Source, @null))
355     {
356         SourcesTreeMethods.Detach(ref
            ↳ GetLinkIndexPartReference(link.Source).RootAsSource, linkIndex);
357     }
358     if (!AreEqual(link.Target, @null))
359     {
360         TargetsTreeMethods.Detach(ref
            ↳ GetLinkIndexPartReference(link.Target).RootAsTarget, linkIndex);
361     }
362     link.Source = substitution[constants.SourcePart];
363     link.Target = substitution[constants.TargetPart];
364     if (!AreEqual(link.Source, @null))
365     {
366         SourcesTreeMethods.Attach(ref
            ↳ GetLinkIndexPartReference(link.Source).RootAsSource, linkIndex);
367     }
368     if (!AreEqual(link.Target, @null))
369     {
370         TargetsTreeMethods.Attach(ref
            ↳ GetLinkIndexPartReference(link.Target).RootAsTarget, linkIndex);
371     }
372     return linkIndex;
373 }
374
375 /// <remarks>
376 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
    ↳ пространство
377 /// </remarks>
378 [MethodImpl(MethodImplOptions.AggressiveInlining)]
379 public virtual TLink Create(IList<TLink> restrictions)
380 {
381     ref var header = ref GetHeaderReference();
382     var freeLink = header.FirstFreeLink;
383     if (!AreEqual(freeLink, Constants.Null))
384     {
385         UnusedLinksListMethods.Detach(freeLink);
386     }
387     else
388     {
389         var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
390         if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
391         {
392             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
393         }
394         if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))

```



```

395     {
396         _dataMemory.ReservedCapacity += _dataMemoryReservationStepInBytes;
397         _indexMemory.ReservedCapacity += _indexMemoryReservationStepInBytes;
398         SetPointers(_dataMemory, _indexMemory);
399         header = ref GetHeaderReference();
400         header.ReservedLinks = ConvertToAddress(_dataMemory.ReservedCapacity /
            ↳ LinkDataPartSizeInBytes);
401     }
402     header.AllocatedLinks = Increment(header.AllocatedLinks);
403     _dataMemory.UsedCapacity += LinkDataPartSizeInBytes;
404     _indexMemory.UsedCapacity += LinkIndexPartSizeInBytes;
405     freeLink = header.AllocatedLinks;
406 }
407 return freeLink;
408 }
409
410 [MethodImpl(MethodImplOptions.AggressiveInlining)]
411 public virtual void Delete(ICollection<TLink> restrictions)
412 {
413     ref var header = ref GetHeaderReference();
414     var link = restrictions[Constants.IndexPart];
415     if (LessThan(link, header.AllocatedLinks))
416     {
417         UnusedLinksListMethods.AttachAsFirst(link);
418     }
419     else if (AreEqual(link, header.AllocatedLinks))
420     {
421         header.AllocatedLinks = Decrement(header.AllocatedLinks);
422         _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
423         _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
424         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
425         ↳ пока не дойдём до первой существующей связи
426         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
427         while (GreaterThan(header.AllocatedLinks, GetZero()) &&
            ↳ IsUnusedLink(header.AllocatedLinks))
428         {
429             UnusedLinksListMethods.Detach(header.AllocatedLinks);
430             header.AllocatedLinks = Decrement(header.AllocatedLinks);
431             _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
432             _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
433         }
434     }
435 }
436
437 [MethodImpl(MethodImplOptions.AggressiveInlining)]
438 public IList<TLink> GetLinkStruct(TLink linkIndex)
439 {
440     ref var link = ref GetLinkDataPartReference(linkIndex);
441     return new Link<TLink>(linkIndex, link.Source, link.Target);
442 }
443
444 /// <remarks>
445 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
446 ↳ адрес реально поменялся
447 ///
448 /// Указатель this.links может быть в том же месте,
449 /// так как 0-я связь не используется и имеет такой же размер как Header,
450 /// поэтому header размещается в том же месте, что и 0-я связь
451 /// </remarks>
452 [MethodImpl(MethodImplOptions.AggressiveInlining)]
453 protected abstract void SetPointers(IResizableDirectMemory dataMemory,
454     ↳ IResizableDirectMemory indexMemory);
455
456 [MethodImpl(MethodImplOptions.AggressiveInlining)]
457 protected virtual void ResetPointers()
458 {
459     SourcesTreeMethods = null;
460     TargetsTreeMethods = null;
461     UnusedLinksListMethods = null;
462 }
463
464 [MethodImpl(MethodImplOptions.AggressiveInlining)]
465 protected abstract ref LinksHeaderIndexPart<TLink> GetHeaderReference();
466
467 [MethodImpl(MethodImplOptions.AggressiveInlining)]
468 protected abstract ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex);
469
470 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

468     protected abstract ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
469         ↪ linkIndex);
470
471     [MethodImpl(MethodImplOptions.AggressiveInlining)]
472     protected virtual bool Exists(TLink link)
473         => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
474         && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
475         && !IsUnusedLink(link);
476
477     [MethodImpl(MethodImplOptions.AggressiveInlining)]
478     protected virtual bool IsUnusedLink(TLink linkIndex)
479     {
480         if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
481             ↪ is not needed
482         {
483             // TODO: Reduce access to memory in different location (should be enough to use
484             ↪ just linkIndexPart)
485             ref var linkDataPart = ref GetLinkDataPartReference(linkIndex);
486             ref var linkIndexPart = ref GetLinkIndexPartReference(linkIndex);
487             return AreEqual(linkIndexPart.SizeAsSource, default) &&
488                 ↪ !AreEqual(linkDataPart.Source, default);
489         }
490         else
491         {
492             return true;
493         }
494     }
495
496     [MethodImpl(MethodImplOptions.AggressiveInlining)]
497     protected virtual TLink GetOne() => _one;
498
499     [MethodImpl(MethodImplOptions.AggressiveInlining)]
500     protected virtual TLink GetZero() => default;
501
502     [MethodImpl(MethodImplOptions.AggressiveInlining)]
503     protected virtual bool AreEqual(TLink first, TLink second) =>
504         ↪ _equalityComparer.Equals(first, second);
505
506     [MethodImpl(MethodImplOptions.AggressiveInlining)]
507     protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
508         ↪ second) < 0;
509
510     [MethodImpl(MethodImplOptions.AggressiveInlining)]
511     protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
512         ↪ _comparer.Compare(first, second) <= 0;
513
514     [MethodImpl(MethodImplOptions.AggressiveInlining)]
515     protected virtual bool GreaterThan(TLink first, TLink second) =>
516         ↪ _comparer.Compare(first, second) > 0;
517
518     [MethodImpl(MethodImplOptions.AggressiveInlining)]
519     protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
520         ↪ _comparer.Compare(first, second) >= 0;
521
522     [MethodImpl(MethodImplOptions.AggressiveInlining)]
523     protected virtual long ConvertToInt64(TLink value) =>
524         ↪ _addressToInt64Converter.Convert(value);
525
526     [MethodImpl(MethodImplOptions.AggressiveInlining)]
527     protected virtual TLink ConvertToAddress(long value) =>
528         ↪ _int64ToAddressConverter.Convert(value);
529
530     [MethodImpl(MethodImplOptions.AggressiveInlining)]
531     protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
532         ↪ second);
533
534     [MethodImpl(MethodImplOptions.AggressiveInlining)]
535     protected virtual TLink Subtract(TLink first, TLink second) =>
536         ↪ Arithmetic<TLink>.Subtract(first, second);
537
538     [MethodImpl(MethodImplOptions.AggressiveInlining)]
539     protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
540
541     [MethodImpl(MethodImplOptions.AggressiveInlining)]
542     protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
543
544     #region Disposable

```

```

533     protected override bool AllowMultipleDisposeCalls
534     {
535         [MethodImpl(MethodImplOptions.AggressiveInlining)]
536         get => true;
537     }
538
539     [MethodImpl(MethodImplOptions.AggressiveInlining)]
540     protected override void Dispose(bool manual, bool wasDisposed)
541     {
542         if (!wasDisposed)
543         {
544             ResetPointers();
545             _dataMemory.DisposeIfPossible();
546             _indexMemory.DisposeIfPossible();
547         }
548     }
549
550     #endregion
551 }
552 }

```

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

```

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

```

```

50     protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
        ↳ element;
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected override void SetPrevious(TLink element, TLink previous) =>
        ↳ GetLinkDataPartReference(element).Source = previous;
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override void SetNext(TLink element, TLink next) =>
        ↳ GetLinkDataPartReference(element).Target = next;
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
60 }
61 }

```

1.33 ./csharp/Platform.Data.Doublets/Memory/Split/LinksHeaderIndexPart.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Unsafe;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Memory.Split
9  {
10     public struct LinksHeaderIndexPart<TLink> : IEquatable<LinksHeaderIndexPart<TLink>>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
13
14         public static readonly long SizeInBytes = Structure<LinksHeaderIndexPart<TLink>>.Size;
15
16         public TLink AllocatedLinks;
17         public TLink ReservedLinks;
18         public TLink FreeLinks;
19         public TLink FirstFreeLink;
20         public TLink LastFreeLink;
21         public TLink RootAsSource;
22         public TLink RootAsTarget;
23         public TLink Reserved8;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public override bool Equals(object obj) => obj is LinksHeaderIndexPart<TLink>
            ↳ linksHeader ? Equals(linksHeader) : false;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public bool Equals(LinksHeaderIndexPart<TLink> other)
30             => _equalityComparer.Equals(AllocatedLinks, other.AllocatedLinks)
31             && _equalityComparer.Equals(ReservedLinks, other.ReservedLinks)
32             && _equalityComparer.Equals(FreeLinks, other.FreeLinks)
33             && _equalityComparer.Equals(FirstFreeLink, other.FirstFreeLink)
34             && _equalityComparer.Equals(LastFreeLink, other.LastFreeLink)
35             && _equalityComparer.Equals(RootAsSource, other.RootAsSource)
36             && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
37             && _equalityComparer.Equals(Reserved8, other.Reserved8);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public override int GetHashCode() => (AllocatedLinks, ReservedLinks, FreeLinks,
            ↳ FirstFreeLink, LastFreeLink, RootAsSource, RootAsTarget, Reserved8).GetHashCode();
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         public static bool operator ==(LinksHeaderIndexPart<TLink> left,
            ↳ LinksHeaderIndexPart<TLink> right) => left.Equals(right);
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public static bool operator !=(LinksHeaderIndexPart<TLink> left,
            ↳ LinksHeaderIndexPart<TLink> right) => !(left == right);
47     }
48 }

```

1.34 ./csharp/Platform.Data.Doublets/Memory/Split/RawLinkDataPart.cs

```

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

```

```

9  {
10 public struct RawLinkDataPart<TLink> : IEquatable<RawLinkDataPart<TLink>>
11 {
12     private static readonly EqualityComparer<TLink> _equalityComparer =
13         ↪ EqualityComparer<TLink>.Default;
14
15     public static readonly long SizeInBytes = Structure<RawLinkDataPart<TLink>>.Size;
16
17     public TLink Source;
18     public TLink Target;
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     public override bool Equals(object obj) => obj is RawLinkDataPart<TLink> link ?
22         ↪ Equals(link) : false;
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     public bool Equals(RawLinkDataPart<TLink> other)
26         => _equalityComparer.Equals(Source, other.Source)
27         && _equalityComparer.Equals(Target, other.Target);
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public override int GetHashCode() => (Source, Target).GetHashCode();
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     public static bool operator ==(RawLinkDataPart<TLink> left, RawLinkDataPart<TLink>
34         ↪ right) => left.Equals(right);
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     public static bool operator !=(RawLinkDataPart<TLink> left, RawLinkDataPart<TLink>
38         ↪ right) => !(left == right);
39 }
40 }

```

1.35 ./csharp/Platform.Data.Doublets/Memory/Split/RawLinkIndexPart.cs

```

1  using Platform.Unsafe;
2  using System;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Memory.Split
9  {
10     public struct RawLinkIndexPart<TLink> : IEquatable<RawLinkIndexPart<TLink>>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↪ EqualityComparer<TLink>.Default;
14
15         public static readonly long SizeInBytes = Structure<RawLinkIndexPart<TLink>>.Size;
16
17         public TLink RootAsSource;
18         public TLink LeftAsSource;
19         public TLink RightAsSource;
20         public TLink SizeAsSource;
21         public TLink RootAsTarget;
22         public TLink LeftAsTarget;
23         public TLink RightAsTarget;
24         public TLink SizeAsTarget;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public override bool Equals(object obj) => obj is RawLinkIndexPart<TLink> link ?
28             ↪ Equals(link) : false;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public bool Equals(RawLinkIndexPart<TLink> other)
32             => _equalityComparer.Equals(RootAsSource, other.RootAsSource)
33             && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
34             && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
35             && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
36             && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
37             && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
38             && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
39             && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public override int GetHashCode() => (RootAsSource, LeftAsSource, RightAsSource,
43             ↪ SizeAsSource, RootAsTarget, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public static bool operator ==(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
47             ↪ right) => left.Equals(right);
48
49     }
50 }

```

```

44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     public static bool operator !=(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
46         ↪ right) => !(left == right);
47 }
48 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Reflection;
3  using Platform.Converters;
4  using Platform.Numbers;
5  using System.Runtime.CompilerServices;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
12         ↪ IConverter<TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↪ EqualityComparer<TLink>.Default;
16         private static readonly TLink _zero = default;
17         private static readonly TLink _one = Arithmetic.Increment(_zero);
18
19         private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
23             ↪ powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
24             ↪ powerOf2ToUnaryNumberConverter;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Convert(TLink number)
28         {
29             var links = _links;
30             var nullConstant = links.Constants.Null;
31             var target = nullConstant;
32             for (var i = 0; !_equalityComparer.Equals(number, _zero) && i <
33                 ↪ NumericType<TLink>.BitsSize; i++)
34             {
35                 if (_equalityComparer.Equals(Bit.And(number, _one), _one))
36                 {
37                     target = _equalityComparer.Equals(target, nullConstant)
38                         ? _powerOf2ToUnaryNumberConverter.Convert(i)
39                         : links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
40                 }
41                 number = Bit.ShiftRight(number, 1);
42             }
43             return target;
44         }
45     }
46 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Converters;
5  using System.Runtime.CompilerServices;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
12         ↪ IConverter<Doublet<TLink>, TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↪ EqualityComparer<TLink>.Default;
16
17         private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
18         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public LinkToItsFrequencyNumberConveter(
22             ILinks<TLink> links,
23             IProperty<TLink, TLink> frequencyPropertyOperator,
24             IConverter<TLink> unaryNumberToAddressConverter)

```

```

23         : base(links)
24     {
25         _frequencyPropertyOperator = frequencyPropertyOperator;
26         _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
27     }
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public TLink Convert(Doublet<TLink> doublet)
31     {
32         var links = _links;
33         var link = links.SearchOrDefault(doublet.Source, doublet.Target);
34         if (_equalityComparer.Equals(link, default))
35         {
36             throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
37         }
38         var frequency = _frequencyPropertyOperator.Get(link);
39         if (_equalityComparer.Equals(frequency, default))
40         {
41             return default;
42         }
43         var frequencyNumber = links.GetSource(frequency);
44         return _unaryNumberToAddressConverter.Convert(frequencyNumber);
45     }
46 }
47 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Exceptions;
3  using Platform.Ranges;
4  using Platform.Converters;
5  using System.Runtime.CompilerServices;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
12         ↪ IConverter<int, TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↪ EqualityComparer<TLink>.Default;
16
17         private readonly TLink[] _unaryNumberPowersOf2;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
21         {
22             _unaryNumberPowersOf2 = new TLink[64];
23             _unaryNumberPowersOf2[0] = one;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Convert(int power)
28         {
29             Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
30                 ↪ - 1), nameof(power));
31             if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
32             {
33                 return _unaryNumberPowersOf2[power];
34             }
35             var previousPowerOf2 = Convert(power - 1);
36             var powerOf2 = _links.GetOrCreate(previousPowerOf2, previousPowerOf2);
37             _unaryNumberPowersOf2[power] = powerOf2;
38             return powerOf2;
39         }
40     }
41 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Converters;
4  using Platform.Numbers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Numbers.Unary
9  {

```

```

10 public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>,
    ↳ IConverter<TLink>
11 {
12     private static readonly EqualityComparer<TLink> _equalityComparer =
        ↳ EqualityComparer<TLink>.Default;
13     private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
        ↳ UncheckedConverter<TLink, ulong>.Default;
14     private static readonly UncheckedConverter<ulong, TLink> _uint64ToAddressConverter =
        ↳ UncheckedConverter<ulong, TLink>.Default;
15     private static readonly TLink _zero = default;
16     private static readonly TLink _one = Arithmetic.Increment(_zero);
17
18     private readonly Dictionary<TLink, TLink> _unaryToUInt64;
19     private readonly TLink _unaryOne;
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
23         : base(links)
24     {
25         _unaryOne = unaryOne;
26         _unaryToUInt64 = CreateUnaryToUInt64Dictionary(links, unaryOne);
27     }
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public TLink Convert(TLink unaryNumber)
31     {
32         if (_equalityComparer.Equals(unaryNumber, default))
33         {
34             return default;
35         }
36         if (_equalityComparer.Equals(unaryNumber, _unaryOne))
37         {
38             return _one;
39         }
40         var links = _links;
41         var source = links.GetSource(unaryNumber);
42         var target = links.GetTarget(unaryNumber);
43         if (_equalityComparer.Equals(source, target))
44         {
45             return _unaryToUInt64[unaryNumber];
46         }
47         else
48         {
49             var result = _unaryToUInt64[source];
50             TLink lastValue;
51             while (!_unaryToUInt64.TryGetValue(target, out lastValue))
52             {
53                 source = links.GetSource(target);
54                 result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
55                 target = links.GetTarget(target);
56             }
57             result = Arithmetic<TLink>.Add(result, lastValue);
58             return result;
59         }
60     }
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     private static Dictionary<TLink, TLink> CreateUnaryToUInt64Dictionary(ILinks<TLink>
        ↳ links, TLink unaryOne)
64     {
65         var unaryToUInt64 = new Dictionary<TLink, TLink>
66         {
67             { unaryOne, _one }
68         };
69         var unary = unaryOne;
70         var number = _one;
71         for (var i = 1; i < 64; i++)
72         {
73             unary = links.GetOrCreate(unary, unary);
74             number = Double(number);
75             unaryToUInt64.Add(unary, number);
76         }
77         return unaryToUInt64;
78     }
79
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     private static TLink Double(TLink number) =>
        ↳ _uint64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
82 }

```



```
83 }
```

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

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Reflection;
4 using Platform.Converters;
5 using Platform.Numbers;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
12         ⇨ IConverter<TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ⇨ EqualityComparer<TLink>.Default;
16         private static readonly TLink _zero = default;
17         private static readonly TLink _one = Arithmetic.Increment(_zero);
18
19         private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
23             ⇨ TLink> powerOf2ToUnaryNumberConverter) : base(links) => _unaryNumberPowerOf2Indicies
24             ⇨ = CreateUnaryNumberPowerOf2IndiciesDictionary(powerOf2ToUnaryNumberConverter);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Convert(TLink sourceNumber)
28         {
29             var links = _links;
30             var nullConstant = links.Constants.Null;
31             var source = sourceNumber;
32             var target = nullConstant;
33             if (!_equalityComparer.Equals(source, nullConstant))
34             {
35                 while (true)
36                 {
37                     if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
38                     {
39                         SetBit(ref target, powerOf2Index);
40                         break;
41                     }
42                     else
43                     {
44                         powerOf2Index = _unaryNumberPowerOf2Indicies[links.GetSource(source)];
45                         SetBit(ref target, powerOf2Index);
46                         source = links.GetTarget(source);
47                     }
48                 }
49             }
50             return target;
51         }
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         private static Dictionary<TLink, int>
55             ⇨ CreateUnaryNumberPowerOf2IndiciesDictionary(IConverter<int, TLink>
56             ⇨ powerOf2ToUnaryNumberConverter)
57         {
58             var unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
59             for (int i = 0; i < NumericType<TLink>.BitsSize; i++)
60             {
61                 unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
62             }
63             return unaryNumberPowerOf2Indicies;
64         }
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         private static void SetBit(ref TLink target, int powerOf2Index) => target =
68             ⇨ Bit.Or(target, Bit.ShiftLeft(_one, powerOf2Index));
69     }
70 }
```

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

```
1 using System.Linq;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Interfaces;
5
```

```

6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.PropertyOperators
9  {
10     public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>, IProperties<TLink, TLink,
        ↪ TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public PropertiesOperator(ILinks<TLink> links) : base(links) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public TLink GetValue(TLink @object, TLink property)
19         {
20             var links = _links;
21             var objectProperty = links.SearchOrDefault(@object, property);
22             if (_equalityComparer.Equals(objectProperty, default))
23             {
24                 return default;
25             }
26             var constants = links.Constants;
27             var valueLink = links.All(constants.Any, objectProperty).SingleOrDefault();
28             if (valueLink == null)
29             {
30                 return default;
31             }
32             return links.GetTarget(valueLink[constants.IndexPart]);
33         }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public void SetValue(TLink @object, TLink property, TLink value)
37         {
38             var links = _links;
39             var objectProperty = links.GetOrCreate(@object, property);
40             links.DeleteMany(links.AllIndices(links.Constants.Any, objectProperty));
41             links.GetOrCreate(objectProperty, value);
42         }
43     }
44 }

```

1.42 ./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs

```

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

```

```

36         return valueContainer;
37     }
38     var links = _links;
39     var constants = links.Constants;
40     var countinueConstant = constants.Continue;
41     var breakConstant = constants.Break;
42     var anyConstant = constants.Any;
43     var query = new Link<TLink>(anyConstant, property, anyConstant);
44     links.Each(candidate =>
45     {
46         var candidateTarget = links.GetTarget(candidate);
47         var valueTarget = links.GetTarget(candidateTarget);
48         if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
49         {
50             valueContainer = links.GetIndex(candidate);
51             return breakConstant;
52         }
53         return countinueConstant;
54     }, query);
55     return valueContainer;
56 }
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
    ↪ ? default : _links.GetTarget(container);
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 public void Set(TLink link, TLink value)
63 {
64     var links = _links;
65     var property = links.GetOrCreate(link, _propertyMarker);
66     var container = GetContainer(property);
67     if (_equalityComparer.Equals(container, default))
68     {
69         links.GetOrCreate(property, value);
70     }
71     else
72     {
73         links.Update(container, property, value);
74     }
75 }
76 }
77 }

```

1.43 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksAvlBalancedTreeMethodsBase.cs

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7  using Platform.Numbers;
8  using static System.Runtime.CompilerServices.Unsafe;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
13 {
14     public unsafe abstract class LinksAvlBalancedTreeMethodsBase<TLink> :
15     ↪ SizedAndThreadedAVLBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
16     {
17         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
18         ↪ UncheckedConverter<TLink, long>.Default;
19         private static readonly UncheckedConverter<TLink, int> _addressToInt32Converter =
20         ↪ UncheckedConverter<TLink, int>.Default;
21         private static readonly UncheckedConverter<bool, TLink> _boolToAddressConverter =
22         ↪ UncheckedConverter<bool, TLink>.Default;
23         private static readonly UncheckedConverter<TLink, bool> _addressToBoolConverter =
24         ↪ UncheckedConverter<TLink, bool>.Default;
25         private static readonly UncheckedConverter<int, TLink> _int32ToAddressConverter =
26         ↪ UncheckedConverter<int, TLink>.Default;
27
28         protected readonly TLink Break;
29         protected readonly TLink Continue;
30         protected readonly byte* Links;
31         protected readonly byte* Header;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
35         ↪ byte* header)
36     {
37
38     }
39 }

```

```

30     Links = links;
31     Header = header;
32     Break = constants.Break;
33     Continue = constants.Continue;
34 }
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 protected abstract TLink GetTreeRoot();
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 protected abstract TLink GetBasePartValue(TLink link);
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
    ↪ rootSource, TLink rootTarget);
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
    ↪ rootSource, TLink rootTarget);
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
    ↪ AsRef<LinksHeader<TLink>>(Header);
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
    ↪ AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
    ↪ _addressToInt64Converter.Convert(link)));
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
56 {
57     ref var link = ref GetLinkReference(linkIndex);
58     return new Link<TLink>(linkIndex, link.Source, link.Target);
59 }
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
63 {
64     ref var firstLink = ref GetLinkReference(first);
65     ref var secondLink = ref GetLinkReference(second);
66     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
67 }
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
71 {
72     ref var firstLink = ref GetLinkReference(first);
73     ref var secondLink = ref GetLinkReference(second);
74     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
75 }
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected virtual TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5,
    ↪ -5);
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 protected virtual void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
    ↪ Bit<TLink>.PartialWrite(storedValue, size, 5, -5);
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 protected virtual bool GetLeftIsChildValue(TLink value)
85 {
86     unchecked
87     {
88         return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 4, 1));
89         //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
90     }
91 }
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 protected virtual void SetLeftIsChildValue(ref TLink storedValue, bool value)
95 {
96     unchecked
97     {
98         var previousValue = storedValue;

```

```

99         var modified = Bit<TLink>.PartialWrite(previousValue,
100             ↪ _boolToAddressConverter.Convert(value), 4, 1);
101         storedValue = modified;
102     }
103 }
104 [MethodImpl(MethodImplOptions.AggressiveInlining)]
105 protected virtual bool GetRightIsChildValue(TLink value)
106 {
107     unchecked
108     {
109         return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 3, 1));
110         //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
111     }
112 }
113
114 [MethodImpl(MethodImplOptions.AggressiveInlining)]
115 protected virtual void SetRightIsChildValue(ref TLink storedValue, bool value)
116 {
117     unchecked
118     {
119         var previousValue = storedValue;
120         var modified = Bit<TLink>.PartialWrite(previousValue,
121             ↪ _boolToAddressConverter.Convert(value), 3, 1);
122         storedValue = modified;
123     }
124 }
125
126 [MethodImpl(MethodImplOptions.AggressiveInlining)]
127 protected bool IsChild(TLink parent, TLink possibleChild)
128 {
129     var parentSize = GetSize(parent);
130     var childSize = GetSizeOrZero(possibleChild);
131     return GreaterThanZero(childSize) && LessOrEqualThan(childSize, parentSize);
132 }
133
134 [MethodImpl(MethodImplOptions.AggressiveInlining)]
135 protected virtual sbyte GetBalanceValue(TLink storedValue)
136 {
137     unchecked
138     {
139         var value = _addressToInt32Converter.Convert(Bit<TLink>.PartialRead(storedValue,
140             ↪ 0, 3));
141         value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
142             ↪ end of sbyte
143         return (sbyte)value;
144     }
145 }
146
147 [MethodImpl(MethodImplOptions.AggressiveInlining)]
148 protected virtual void SetBalanceValue(ref TLink storedValue, sbyte value)
149 {
150     unchecked
151     {
152         var packagedValue = _int32ToAddressConverter.Convert((byte)value >> 5 & 4 |
153             ↪ value & 3);
154         var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
155         storedValue = modified;
156     }
157 }
158
159 public TLink this[TLink index]
160 {
161     [MethodImpl(MethodImplOptions.AggressiveInlining)]
162     get
163     {
164         var root = GetTreeRoot();
165         if (GreaterOrEqualThan(index, GetSize(root)))
166         {
167             return Zero;
168         }
169         while (!EqualToZero(root))
170         {
171             var left = GetLeftOrDefault(root);
172             var leftSize = GetSizeOrZero(left);
173             if (LessThan(index, leftSize))
174             {
175                 root = left;
176                 continue;
177             }

```

```

173     }
174     if (AreEqual(index, leftSize))
175     {
176         return root;
177     }
178     root = GetRightOrDefault(root);
179     index = Subtract(index, Increment(leftSize));
180 }
181 return Zero; // TODO: Impossible situation exception (only if tree structure
    ↳ broken)
182 }
183 }
184
185 /// <summary>
186 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
    ↳ (концом).
187 /// </summary>
188 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
189 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
190 /// <returns>Индекс искомой связи.</returns>
191 [MethodImpl(MethodImplOptions.AggressiveInlining)]
192 public TLink Search(TLink source, TLink target)
193 {
194     var root = GetTreeRoot();
195     while (!EqualToZero(root))
196     {
197         ref var rootLink = ref GetLinkReference(root);
198         var rootSource = rootLink.Source;
199         var rootTarget = rootLink.Target;
200         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
            ↳ node.Key < root.Key
201         {
202             root = GetLeftOrDefault(root);
203         }
204         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
            ↳ node.Key > root.Key
205         {
206             root = GetRightOrDefault(root);
207         }
208         else // node.Key == root.Key
209         {
210             return root;
211         }
212     }
213     return Zero;
214 }
215
216 // TODO: Return indices range instead of references count
217 [MethodImpl(MethodImplOptions.AggressiveInlining)]
218 public TLink CountUsages(TLink link)
219 {
220     var root = GetTreeRoot();
221     var total = GetSize(root);
222     var totalRightIgnore = Zero;
223     while (!EqualToZero(root))
224     {
225         var @base = GetBasePartValue(root);
226         if (LessOrEqualThan(@base, link))
227         {
228             root = GetRightOrDefault(root);
229         }
230         else
231         {
232             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
233             root = GetLeftOrDefault(root);
234         }
235     }
236     root = GetTreeRoot();
237     var totalLeftIgnore = Zero;
238     while (!EqualToZero(root))
239     {
240         var @base = GetBasePartValue(root);
241         if (GreaterOrEqualThan(@base, link))
242         {
243             root = GetLeftOrDefault(root);
244         }
245         else
246         {

```

```

247         totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
248
249         root = GetRightOrDefault(root);
250     }
251 }
252 return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
253 }
254
255 [MethodImpl(MethodImplOptions.AggressiveInlining)]
256 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
257 {
258     var root = GetTreeRoot();
259     if (EqualToZero(root))
260     {
261         return Continue;
262     }
263     TLink first = Zero, current = root;
264     while (!EqualToZero(current))
265     {
266         var @base = GetBasePartValue(current);
267         if (GreaterOrEqualThan(@base, link))
268         {
269             if (AreEqual(@base, link))
270             {
271                 first = current;
272             }
273             current = GetLeftOrDefault(current);
274         }
275         else
276         {
277             current = GetRightOrDefault(current);
278         }
279     }
280     if (!EqualToZero(first))
281     {
282         current = first;
283         while (true)
284         {
285             if (AreEqual(handler(GetLinkValues(current)), Break))
286             {
287                 return Break;
288             }
289             current = GetNext(current);
290             if (EqualToZero(current) || !AreEqual(GetBasePartValue(current), link))
291             {
292                 break;
293             }
294         }
295     }
296     return Continue;
297 }
298
299 [MethodImpl(MethodImplOptions.AggressiveInlining)]
300 protected override void PrintNodeValue(TLink node, StringBuilder sb)
301 {
302     ref var link = ref GetLinkReference(node);
303     sb.Append(' ');
304     sb.Append(link.Source);
305     sb.Append('-');
306     sb.Append('>');
307     sb.Append(link.Target);
308 }
309 }
310 }

```

1.44 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSizeBalancedTreeMethodsBase.cs

```

1 using System;
2 using System.Text;
3 using System.Collections.Generic;
4 using System.Runtime.CompilerServices;
5 using Platform.Collections.Methods.Trees;
6 using Platform.Converters;
7 using static System.Runtime.CompilerServices.Unsafe;
8
9 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
12 {
13     public unsafe abstract class LinksSizeBalancedTreeMethodsBase<TLink> :
14         ↳ SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>

```

```

14 {
15     private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
16         ↳ UncheckedConverter<TLink, long>.Default;
17
18     protected readonly TLink Break;
19     protected readonly TLink Continue;
20     protected readonly byte* Links;
21     protected readonly byte* Header;
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
25         ↳ byte* header)
26     {
27         Links = links;
28         Header = header;
29         Break = constants.Break;
30         Continue = constants.Continue;
31     }
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected abstract TLink GetTreeRoot();
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected abstract TLink GetBasePartValue(TLink link);
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
41         ↳ rootSource, TLink rootTarget);
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
45         ↳ rootSource, TLink rootTarget);
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
49         ↳ AsRef<LinksHeader<TLink>>(Header);
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
53         ↳ AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
54         ↳ _addressToInt64Converter.Convert(link)));
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
58     {
59         ref var link = ref GetLinkReference(linkIndex);
60         return new Link<TLink>(linkIndex, link.Source, link.Target);
61     }
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
65     {
66         ref var firstLink = ref GetLinkReference(first);
67         ref var secondLink = ref GetLinkReference(second);
68         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
69             ↳ secondLink.Source, secondLink.Target);
70     }
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
74     {
75         ref var firstLink = ref GetLinkReference(first);
76         ref var secondLink = ref GetLinkReference(second);
77         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
78             ↳ secondLink.Source, secondLink.Target);
79     }
80
81     public TLink this[TLink index]
82     {
83         [MethodImpl(MethodImplOptions.AggressiveInlining)]
84         get
85         {
86             var root = GetTreeRoot();
87             if (GreaterOrEqualThan(index, GetSize(root)))
88             {
89                 return Zero;
90             }
91             while (!EqualToZero(root))
92             {

```



```

84         var left = GetLeftOrDefault(root);
85         var leftSize = GetSizeOrZero(left);
86         if (LessThan(index, leftSize))
87         {
88             root = left;
89             continue;
90         }
91         if (AreEqual(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 /// <summary>
103 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
104 ↪ (концом).
105 /// </summary>
106 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
107 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
108 /// <returns>Индекс искомой связи.</returns>
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
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;
118         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
119             ↪ node.Key < root.Key
120         {
121             root = GetLeftOrDefault(root);
122         }
123         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
124             ↪ node.Key > root.Key
125         {
126             root = GetRightOrDefault(root);
127         }
128         else // node.Key == root.Key
129         {
130             return root;
131         }
132     }
133     return Zero;
134 }
135
136 // TODO: Return indices range instead of references count
137 [MethodImpl(MethodImplOptions.AggressiveInlining)]
138 public TLink CountUsages(TLink link)
139 {
140     var root = GetTreeRoot();
141     var total = GetSize(root);
142     var totalRightIgnore = Zero;
143     while (!EqualToZero(root))
144     {
145         var @base = GetBasePartValue(root);
146         if (LessOrEqualThan(@base, link))
147         {
148             root = GetRightOrDefault(root);
149         }
150         else
151         {
152             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
153             root = GetLeftOrDefault(root);
154         }
155     }
156     root = GetTreeRoot();
157     var totalLeftIgnore = Zero;
158     while (!EqualToZero(root))
159     {
160         var @base = GetBasePartValue(root);

```

```

158         if (GreaterOrEqualThan(@base, link))
159         {
160             root = GetLeftOrDefault(root);
161         }
162         else
163         {
164             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
165             root = GetRightOrDefault(root);
166         }
167     }
168     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
169 }
170
171 [MethodImpl(MethodImplOptions.AggressiveInlining)]
172 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
173     ↳ EachUsageCore(@base, GetTreeRoot(), handler);
174
175 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
176 ↳ low-level MSIL stack.
177 [MethodImpl(MethodImplOptions.AggressiveInlining)]
178 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
179 {
180     var @continue = Continue;
181     if (EqualToZero(link))
182     {
183         return @continue;
184     }
185     var linkBasePart = GetBasePartValue(link);
186     var @break = Break;
187     if (GreaterThan(linkBasePart, @base))
188     {
189         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
190         {
191             return @break;
192         }
193     }
194     else if (LessThan(linkBasePart, @base))
195     {
196         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
197         {
198             return @break;
199         }
200     }
201     else //if (linkBasePart == @base)
202     {
203         if (AreEqual(handler(GetLinkValues(link)), @break))
204         {
205             return @break;
206         }
207         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
208         {
209             return @break;
210         }
211         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
212         {
213             return @break;
214         }
215     }
216     return @continue;
217 }
218
219 [MethodImpl(MethodImplOptions.AggressiveInlining)]
220 protected override void PrintNodeValue(TLink node, StringBuilder sb)
221 {
222     ref var link = ref GetLinkReference(node);
223     sb.Append(' ');
224     sb.Append(link.Source);
225     sb.Append(' - ');
226     sb.Append(' > ');
227     sb.Append(link.Target);
228 }

```

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

```

```

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 }
75 }

```

1.46 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesSizeBalancedTreeMethods.cs

```

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

```

```
56     }
57 }
```

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

```

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

```

1.48 ./csharp/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> :
        ↪ LinksSizeBalancedTreeMethodsBase<TLink>
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public LinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
            ↪ byte* header) : base(constants, links, header) { }
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         protected override ref TLink GetLeftReference(TLink node) => ref
            ↪ GetLinkReference(node).LeftAsTarget;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected override ref TLink GetRightReference(TLink node) => ref
            ↪ GetLinkReference(node).RightAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override void SetLeft(TLink node, TLink left) =>
            ↪ GetLinkReference(node).LeftAsTarget = left;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override void SetRight(TLink node, TLink right) =>
            ↪ GetLinkReference(node).RightAsTarget = right;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsTarget;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override void SetSize(TLink node, TLink size) =>
            ↪ GetLinkReference(node).SizeAsTarget = size;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsTarget;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override bool FirstIsToLeftOfSecond(TLink firstSource, TLink firstTarget,
            ↪ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
            ↪ (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
            ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
            ↪ (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
47

```

```

48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected override void ClearNode(TLink node)
50     {
51         ref var link = ref GetLinkReference(node);
52         link.LeftAsTarget = Zero;
53         link.RightAsTarget = Zero;
54         link.SizeAsTarget = Zero;
55     }
56 }
57 }

```

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

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using static System.Runtime.CompilerServices.Unsafe;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
10 {
11     public unsafe class ResizableDirectMemoryLinks<TLink> : ResizableDirectMemoryLinksBase<TLink>
12     {
13         private readonly Func<ILinksTreeMethods<TLink>> _createSourceTreeMethods;
14         private readonly Func<ILinksTreeMethods<TLink>> _createTargetTreeMethods;
15         private byte* _header;
16         private byte* _links;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public ResizableDirectMemoryLinks(string address) : this(address, DefaultLinksSizeStep)
20         { }
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 ResizableDirectMemoryLinks(string address, long memoryReservationStep) : this(new
31         { }
32         { }
33         { }
34         { }
35         { }
36         { }
37         { }
38         { }
39         { }
40         { }
41         { }
42         { }
43         { }
44         { }
45         { }
46         { }
47         { }
48         { }
49         { }
50         { }
51         { }
52         { }
53         { }
54         { }
55         { }
56         { }
57         { }
58         { }
59         { }
60         { }
61         { }
62         { }
63         { }
64         { }
65         { }
66         { }
67         { }
68         { }
69         { }
70         { }
71         { }
72         { }
73         { }
74         { }
75         { }
76         { }
77         { }
78         { }
79         { }
80         { }
81         { }
82         { }
83         { }
84         { }
85         { }
86         { }
87         { }
88         { }
89         { }
90         { }
91         { }
92         { }
93         { }
94         { }
95         { }
96         { }
97         { }
98         { }
99         { }
100        { }
101        { }
102        { }
103        { }
104        { }
105        { }
106        { }
107        { }
108        { }
109        { }
110        { }
111        { }
112        { }
113        { }
114        { }
115        { }
116        { }
117        { }
118        { }
119        { }
120        { }
121        { }
122        { }
123        { }
124        { }
125        { }
126        { }
127        { }
128        { }
129        { }
130        { }
131        { }
132        { }
133        { }
134        { }
135        { }
136        { }
137        { }
138        { }
139        { }
140        { }
141        { }
142        { }
143        { }
144        { }
145        { }
146        { }
147        { }
148        { }
149        { }
150        { }
151        { }
152        { }
153        { }
154        { }
155        { }
156        { }
157        { }
158        { }
159        { }
160        { }
161        { }
162        { }
163        { }
164        { }
165        { }
166        { }
167        { }
168        { }
169        { }
170        { }
171        { }
172        { }
173        { }
174        { }
175        { }
176        { }
177        { }
178        { }
179        { }
180        { }
181        { }
182        { }
183        { }
184        { }
185        { }
186        { }
187        { }
188        { }
189        { }
190        { }
191        { }
192        { }
193        { }
194        { }
195        { }
196        { }
197        { }
198        { }
199        { }
200        { }
201        { }
202        { }
203        { }
204        { }
205        { }
206        { }
207        { }
208        { }
209        { }
210        { }
211        { }
212        { }
213        { }
214        { }
215        { }
216        { }
217        { }
218        { }
219        { }
220        { }
221        { }
222        { }
223        { }
224        { }
225        { }
226        { }
227        { }
228        { }
229        { }
230        { }
231        { }
232        { }
233        { }
234        { }
235        { }
236        { }
237        { }
238        { }
239        { }
240        { }
241        { }
242        { }
243        { }
244        { }
245        { }
246        { }
247        { }
248        { }
249        { }
250        { }
251        { }
252        { }
253        { }
254        { }
255        { }
256        { }
257        { }
258        { }
259        { }
260        { }
261        { }
262        { }
263        { }
264        { }
265        { }
266        { }
267        { }
268        { }
269        { }
270        { }
271        { }
272        { }
273        { }
274        { }
275        { }
276        { }
277        { }
278        { }
279        { }
280        { }
281        { }
282        { }
283        { }
284        { }
285        { }
286        { }
287        { }
288        { }
289        { }
290        { }
291        { }
292        { }
293        { }
294        { }
295        { }
296        { }
297        { }
298        { }
299        { }
300        { }
301        { }
302        { }
303        { }
304        { }
305        { }
306        { }
307        { }
308        { }
309        { }
310        { }
311        { }
312        { }
313        { }
314        { }
315        { }
316        { }
317        { }
318        { }
319        { }
320        { }
321        { }
322        { }
323        { }
324        { }
325        { }
326        { }
327        { }
328        { }
329        { }
330        { }
331        { }
332        { }
333        { }
334        { }
335        { }
336        { }
337        { }
338        { }
339        { }
340        { }
341        { }
342        { }
343        { }
344        { }
345        { }
346        { }
347        { }
348        { }
349        { }
350        { }
351        { }
352        { }
353        { }
354        { }
355        { }
356        { }
357        { }
358        { }
359        { }
360        { }
361        { }
362        { }
363        { }
364        { }
365        { }
366        { }
367        { }
368        { }
369        { }
370        { }
371        { }
372        { }
373        { }
374        { }
375        { }
376        { }
377        { }
378        { }
379        { }
380        { }
381        { }
382        { }
383        { }
384        { }
385        { }
386        { }
387        { }
388        { }
389        { }
390        { }
391        { }
392        { }
393        { }
394        { }
395        { }
396        { }
397        { }
398        { }
399        { }
400        { }
401        { }
402        { }
403        { }
404        { }
405        { }
406        { }
407        { }
408        { }
409        { }
410        { }
411        { }
412        { }
413        { }
414        { }
415        { }
416        { }
417        { }
418        { }
419        { }
420        { }
421        { }
422        { }
423        { }
424        { }
425        { }
426        { }
427        { }
428        { }
429        { }
430        { }
431        { }
432        { }
433        { }
434        { }
435        { }
436        { }
437        { }
438        { }
439        { }
440        { }
441        { }
442        { }
443        { }
444        { }
445        { }
446        { }
447        { }
448        { }
449        { }
450        { }
451        { }
452        { }
453        { }
454        { }
455        { }
456        { }
457        { }
458        { }
459        { }
460        { }
461        { }
462        { }
463        { }
464        { }
465        { }
466        { }
467        { }
468        { }
469        { }
470        { }
471        { }
472        { }
473        { }
474        { }
475        { }
476        { }
477        { }
478        { }
479        { }
480        { }
481        { }
482        { }
483        { }
484        { }
485        { }
486        { }
487        { }
488        { }
489        { }
490        { }
491        { }
492        { }
493        { }
494        { }
495        { }
496        { }
497        { }
498        { }
499        { }
500        { }
501        { }
502        { }
503        { }
504        { }
505        { }
506        { }
507        { }
508        { }
509        { }
510        { }
511        { }
512        { }
513        { }
514        { }
515        { }
516        { }
517        { }
518        { }
519        { }
520        { }
521        { }
522        { }
523        { }
524        { }
525        { }
526        { }
527        { }
528        { }
529        { }
530        { }
531        { }
532        { }
533        { }
534        { }
535        { }
536        { }
537        { }
538        { }
539        { }
540        { }
541        { }
542        { }
543        { }
544        { }
545        { }
546        { }
547        { }
548        { }
549        { }
550        { }
551        { }
552        { }
553        { }
554        { }
555        { }
556        { }
557        { }
558        { }
559        { }
560        { }
561        { }
562        { }
563        { }
564        { }
565        { }
566        { }
567        { }
568        { }
569        { }
570        { }
571        { }
572        { }
573        { }
574        { }
575        { }
576        { }
577        { }
578        { }
579        { }
580        { }
581        { }
582        { }
583        { }
584        { }
585        { }
586        { }
587        { }
588        { }
589        { }
590        { }
591        { }
592        { }
593        { }
594        { }
595        { }
596        { }
597        { }
598        { }
599        { }
600        { }
601        { }
602        { }
603        { }
604        { }
605        { }
606        { }
607        { }
608        { }
609        { }
610        { }
611        { }
612        { }
613        { }
614        { }
615        { }
616        { }
617        { }
618        { }
619        { }
620        { }
621        { }
622        { }
623        { }
624        { }
625        { }
626        { }
627        { }
628        { }
629        { }
630        { }
631        { }
632        { }
633        { }
634        { }
635        { }
636        { }
637        { }
638        { }
639        { }
640        { }
641        { }
642        { }
643        { }
644        { }
645        { }
646        { }
647        { }
648        { }
649        { }
650        { }
651        { }
652        { }
653        { }
654        { }
655        { }
656        { }
657        { }
658        { }
659        { }
660        { }
661        { }
662        { }
663        { }
664        { }
665        { }
666        { }
667        { }
668        { }
669        { }
670        { }
671        { }
672        { }
673        { }
674        { }
675        { }
676        { }
677        { }
678        { }
679        { }
680        { }
681        { }
682        { }
683        { }
684        { }
685        { }
686        { }
687        { }
688        { }
689        { }
690        { }
691        { }
692        { }
693        { }
694        { }
695        { }
696        { }
697        { }
698        { }
699        { }
700        { }
701        { }
702        { }
703        { }
704        { }
705        { }
706        { }
707        { }
708        { }
709        { }
710        { }
711        { }
712        { }
713        { }
714        { }
715        { }
716        { }
717        { }
718        { }
719        { }
720        { }
721        { }
722        { }
723        { }
724        { }
725        { }
726        { }
727        { }
728        { }
729        { }
730        { }
731        { }
732        { }
733        { }
734        { }
735        { }
736        { }
737        { }
738        { }
739        { }
740        { }
741        { }
742        { }
743        { }
744        { }
745        { }
746        { }
747        { }
748        { }
749        { }
750        { }
751        { }
752        { }
753        { }
754        { }
755        { }
756        { }
757        { }
758        { }
759        { }
760        { }
761        { }
762        { }
763        { }
764        { }
765        { }
766        { }
767        { }
768        { }
769        { }
770        { }
771        { }
772        { }
773        { }
774        { }
775        { }
776        { }
777        { }
778        { }
779        { }
780        { }
781        { }
782        { }
783        { }
784        { }
785        { }
786        { }
787        { }
788        { }
789        { }
790        { }
791        { }
792        { }
793        { }
794        { }
795        { }
796        { }
797        { }
798        { }
799        { }
800        { }
801        { }
802        { }
803        { }
804        { }
805        { }
806        { }
807        { }
808        { }
809        { }
810        { }
811        { }
812        { }
813        { }
814        { }
815        { }
816        { }
817        { }
818        { }
819        { }
820        { }
821        { }
822        { }
823        { }
824        { }
825        { }
826        { }
827        { }
828        { }
829        { }
830        { }
831        { }
832        { }
833        { }
834        { }
835        { }
836        { }
837        { }
838        { }
839        { }
840        { }
841        { }
842        { }
843        { }
844        { }
845        { }
846        { }
847        { }
848        { }
849        { }
850        { }
851        { }
852        { }
853        { }
854        { }
855        { }
856        { }
857        { }
858        { }
859        { }
860        { }
861        { }
862        { }
863        { }
864        { }
865        { }
866        { }
867        { }
868        { }
869        { }
870        { }
871        { }
872        { }
873        { }
874        { }
875        { }
876        { }
877        { }
878        { }
879        { }
880        { }
881        { }
882        { }
883        { }
884        { }
885        { }
886        { }
887        { }
888        { }
889        { }
890        { }
891        { }
892        { }
893        { }
894        { }
895        { }
896        { }
897        { }
898        { }
899        { }
900        { }
901        { }
902        { }
903        { }
904        { }
905        { }
906        { }
907        { }
908        { }
909        { }
910        { }
911        { }
912        { }
913        { }
914        { }
915        { }
916        { }
917        { }
918        { }
919        { }
920        { }
921        { }
922        { }
923        { }
924        { }
925        { }
926        { }
927        { }
928        { }
929        { }
930        { }
931        { }
932        { }
933        { }
934        { }
935        { }
936        { }
937        { }
938        { }
939        { }
940        { }
941        { }
942        { }
943        { }
944        { }
945        { }
946        { }
947        { }
948        { }
949        { }
950        { }
951        { }
952        { }
953        { }
954        { }
955        { }
956        { }
957        { }
958        { }
959        { }
960        { }
961        { }
962        { }
963        { }
964        { }
965        { }
966        { }
967        { }
968        { }
969        { }
970        { }
971        { }
972        { }
973        { }
974        { }
975        { }
976        { }
977        { }
978        { }
979        { }
980        { }
981        { }
982        { }
983        { }
984        { }
985        { }
986        { }
987        { }
988        { }
989        { }
990        { }
991        { }
992        { }
993        { }
994        { }
995        { }
996        { }
997        { }
998        { }
999        { }
1000       { }

```

```

54     _links = (byte*)memory.Pointer;
55     _header = _links;
56     SourcesTreeMethods = _createSourceTreeMethods();
57     TargetsTreeMethods = _createTargetTreeMethods();
58     UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_links, _header);
59 }
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected override void ResetPointers()
63 {
64     base.ResetPointers();
65     _links = null;
66     _header = null;
67 }
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected override ref LinksHeader<TLink> GetHeaderReference() => ref
71     ↪ AsRef<LinksHeader<TLink>>(_header);
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 protected override ref RawLink<TLink> GetLinkReference(TLink linkIndex) => ref
75     ↪ AsRef<RawLink<TLink>>(_links + (LinkSizeInBytes * ConvertToInt64(linkIndex)));
76 }
77 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5  using Platform.Singletons;
6  using Platform.Converters;
7  using Platform.Numbers;
8  using Platform.Memory;
9  using Platform.Data.Exceptions;
10
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
14 {
15     public abstract class ResizableDirectMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16     {
17         private static readonly EqualityComparer<TLink> _equalityComparer =
18             ↪ EqualityComparer<TLink>.Default;
19         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
20         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
21             ↪ UncheckedConverter<TLink, long>.Default;
22         private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
23             ↪ UncheckedConverter<long, TLink>.Default;
24
25         private static readonly TLink _zero = default;
26         private static readonly TLink _one = Arithmetic.Increment(_zero);
27
28         /// <summary>Возвращает размер одной связи в байтах.</summary>
29         /// <remarks>
30         /// Используется только во вне класса, не рекомендуется использовать внутри.
31         /// Так как во вне не обязательно будет доступен unsafe C#.
32         /// </remarks>
33         public static readonly long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;
34
35         public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
36
37         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
38
39         protected readonly IResizableDirectMemory _memory;
40         protected readonly long _memoryReservationStep;
41
42         protected ILinksTreeMethods<TLink> TargetsTreeMethods;
43         protected ILinksTreeMethods<TLink> SourcesTreeMethods;
44         // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
45         ↪ нужно использовать не список а дерево, так как так можно быстрее проверить на
46         ↪ наличие связи внутри
47         protected ILinksListMethods<TLink> UnusedLinksListMethods;
48
49         /// <summary>
50         /// Возвращает общее число связей находящихся в хранилище.
51         /// </summary>
52         protected virtual TLink Total
53         {
54             [MethodImpl(MethodImplOptions.AggressiveInlining)]
55             get

```



```

51     {
52         ref var header = ref GetHeaderReference();
53         return Subtract(header.AllocatedLinks, header.FreeLinks);
54     }
55 }
56
57 public virtual LinksConstants<TLink> Constants
58 {
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     get;
61 }
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected ResizableDirectMemoryLinksBase(IResizableDirectMemory memory, long
    ↳ memoryReservationStep, LinksConstants<TLink> constants)
65 {
66     _memory = memory;
67     _memoryReservationStep = memoryReservationStep;
68     Constants = constants;
69 }
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 protected ResizableDirectMemoryLinksBase(IResizableDirectMemory memory, long
    ↳ memoryReservationStep) : this(memory, memoryReservationStep,
    ↳ Default<LinksConstants<TLink>>.Instance) { }
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
76 {
77     if (memory.ReservedCapacity < memoryReservationStep)
78     {
79         memory.ReservedCapacity = memoryReservationStep;
80     }
81     SetPointers(memory);
82     ref var header = ref GetHeaderReference();
83     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
84     memory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) * LinkSizeInBytes) +
    ↳ LinkHeaderSizeInBytes;
85     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
86     header.ReservedLinks = ConvertToAddress((memory.ReservedCapacity -
    ↳ LinkHeaderSizeInBytes) / LinkSizeInBytes);
87 }
88
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 public virtual TLink Count(IList<TLink> restrictions)
91 {
92     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
93     if (restrictions.Count == 0)
94     {
95         return Total;
96     }
97     var constants = Constants;
98     var any = constants.Any;
99     var index = restrictions[constants.IndexPart];
100     if (restrictions.Count == 1)
101     {
102         if (AreEqual(index, any))
103         {
104             return Total;
105         }
106         return Exists(index) ? GetOne() : GetZero();
107     }
108     if (restrictions.Count == 2)
109     {
110         var value = restrictions[1];
111         if (AreEqual(index, any))
112         {
113             if (AreEqual(value, any))
114             {
115                 return Total; // Any - как отсутствие ограничения
116             }
117             return Add(SourcesTreeMethods.CountUsages(value),
    ↳ TargetsTreeMethods.CountUsages(value));
118         }
119         else
120         {
121             if (!Exists(index))
122             {

```

```

123         return GetZero();
124     }
125     if (AreEqual(value, any))
126     {
127         return GetOne();
128     }
129     ref var storedLinkValue = ref GetLinkReference(index);
130     if (AreEqual(storedLinkValue.Source, value) ||
131         ⇨ AreEqual(storedLinkValue.Target, value))
132     {
133         return GetOne();
134     }
135     return GetZero();
136 }
137 if (restrictions.Count == 3)
138 {
139     var source = restrictions[constants.SourcePart];
140     var target = restrictions[constants.TargetPart];
141     if (AreEqual(index, any))
142     {
143         if (AreEqual(source, any) && AreEqual(target, any))
144         {
145             return Total;
146         }
147         else if (AreEqual(source, any))
148         {
149             return TargetsTreeMethods.CountUsages(target);
150         }
151         else if (AreEqual(target, any))
152         {
153             return SourcesTreeMethods.CountUsages(source);
154         }
155         else //if(source != Any && target != Any)
156         {
157             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
158             var link = SourcesTreeMethods.Search(source, target);
159             return AreEqual(link, constants.Null) ? GetZero() : GetOne();
160         }
161     }
162     else
163     {
164         if (!Exists(index))
165         {
166             return GetZero();
167         }
168         if (AreEqual(source, any) && AreEqual(target, any))
169         {
170             return GetOne();
171         }
172         ref var storedLinkValue = ref GetLinkReference(index);
173         if (!AreEqual(source, any) && !AreEqual(target, any))
174         {
175             if (AreEqual(storedLinkValue.Source, source) &&
176                 ⇨ AreEqual(storedLinkValue.Target, target))
177             {
178                 return GetOne();
179             }
180             return GetZero();
181         }
182         var value = default(TLink);
183         if (AreEqual(source, any))
184         {
185             value = target;
186         }
187         if (AreEqual(target, any))
188         {
189             value = source;
190         }
191         if (AreEqual(storedLinkValue.Source, value) ||
192             ⇨ AreEqual(storedLinkValue.Target, value))
193         {
194             return GetOne();
195         }
196         return GetZero();
197     }
198 }

```

```

197         throw new NotSupportedException("Другие размеры и способы ограничений не
198         ↪ поддерживаются.");
199     }
200     [MethodImpl(MethodImplOptions.AggressiveInlining)]
201     public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
202     {
203         var constants = Constants;
204         var @break = constants.Break;
205         if (restrictions.Count == 0)
206         {
207             for (var link = GetOne(); LessOrEqualThan(link,
208                 ↪ GetHeaderReference().AllocatedLinks); link = Increment(link))
209             {
210                 if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
211                 {
212                     return @break;
213                 }
214             }
215             return @break;
216         }
217         var @continue = constants.Continue;
218         var any = constants.Any;
219         var index = restrictions[constants.IndexPart];
220         if (restrictions.Count == 1)
221         {
222             if (AreEqual(index, any))
223             {
224                 return Each(handler, Array.Empty<TLink>());
225             }
226             if (!Exists(index))
227             {
228                 return @continue;
229             }
230             return handler(GetLinkStruct(index));
231         }
232         if (restrictions.Count == 2)
233         {
234             var value = restrictions[1];
235             if (AreEqual(index, any))
236             {
237                 if (AreEqual(value, any))
238                 {
239                     return Each(handler, Array.Empty<TLink>());
240                 }
241                 if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
242                 {
243                     return @break;
244                 }
245                 return Each(handler, new Link<TLink>(index, any, value));
246             }
247             else
248             {
249                 if (!Exists(index))
250                 {
251                     return @continue;
252                 }
253                 if (AreEqual(value, any))
254                 {
255                     return handler(GetLinkStruct(index));
256                 }
257                 ref var storedLinkValue = ref GetLinkReference(index);
258                 if (AreEqual(storedLinkValue.Source, value) ||
259                     AreEqual(storedLinkValue.Target, value))
260                 {
261                     return handler(GetLinkStruct(index));
262                 }
263                 return @continue;
264             }
265         }
266         if (restrictions.Count == 3)
267         {
268             var source = restrictions[constants.SourcePart];
269             var target = restrictions[constants.TargetPart];
270             if (AreEqual(index, any))
271             {
272                 if (AreEqual(source, any) && AreEqual(target, any))
273             
```

```

273         return Each(handler, Array.Empty<TLink>());
274     }
275     else if (AreEqual(source, any))
276     {
277         return TargetsTreeMethods.EachUsage(target, handler);
278     }
279     else if (AreEqual(target, any))
280     {
281         return SourcesTreeMethods.EachUsage(source, handler);
282     }
283     else //if(source != Any && target != Any)
284     {
285         var link = SourcesTreeMethods.Search(source, target);
286         return AreEqual(link, constants.Null) ? @continue :
            ↪ handler(GetLinkStruct(link));
287     }
288 }
289 else
290 {
291     if (!Exists(index))
292     {
293         return @continue;
294     }
295     if (AreEqual(source, any) && AreEqual(target, any))
296     {
297         return handler(GetLinkStruct(index));
298     }
299     ref var storedLinkValue = ref GetLinkReference(index);
300     if (!AreEqual(source, any) && !AreEqual(target, any))
301     {
302         if (AreEqual(storedLinkValue.Source, source) &&
303             AreEqual(storedLinkValue.Target, target))
304         {
305             return handler(GetLinkStruct(index));
306         }
307         return @continue;
308     }
309     var value = default(TLink);
310     if (AreEqual(source, any))
311     {
312         value = target;
313     }
314     if (AreEqual(target, any))
315     {
316         value = source;
317     }
318     if (AreEqual(storedLinkValue.Source, value) ||
319         AreEqual(storedLinkValue.Target, value))
320     {
321         return handler(GetLinkStruct(index));
322     }
323     return @continue;
324 }
325 }
326 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↪ поддерживаются.");
327 }
328
329 /// <remarks>
330 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
331 ↪ в другом месте (но не в менеджере памяти, а в логике Links)
332 /// </remarks>
333 [MethodImpl(MethodImplOptions.AggressiveInlining)]
334 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
335 {
336     var constants = Constants;
337     var @null = constants.Null;
338     var linkIndex = restrictions[constants.IndexPart];
339     ref var link = ref GetLinkReference(linkIndex);
340     ref var header = ref GetHeaderReference();
341     ref var firstAsSource = ref header.FirstAsSource;
342     ref var firstAsTarget = ref header.FirstAsTarget;
343     // Будет корректно работать только в том случае, если пространство выделенной связи
344     ↪ предварительно заполнено нулями
345     if (!AreEqual(link.Source, @null))
346     {
347         SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
348     }
349 }

```

```

347     if (!AreEqual(link.Target, @null))
348     {
349         TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
350     }
351     link.Source = substitution[constants.SourcePart];
352     link.Target = substitution[constants.TargetPart];
353     if (!AreEqual(link.Source, @null))
354     {
355         SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
356     }
357     if (!AreEqual(link.Target, @null))
358     {
359         TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
360     }
361     return linkIndex;
362 }
363
364 /// <remarks>
365 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
366 /// ↪ пространство
367 /// </remarks>
368 [MethodImpl(MethodImplOptions.AggressiveInlining)]
369 public virtual TLink Create(ICollection<TLink> restrictions)
370 {
371     ref var header = ref GetHeaderReference();
372     var freeLink = header.FirstFreeLink;
373     if (!AreEqual(freeLink, Constants.Null))
374     {
375         UnusedLinksListMethods.Detach(freeLink);
376     }
377     else
378     {
379         var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
380         if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
381         {
382             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
383         }
384         if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks))
385         {
386             _memory.ReservedCapacity += _memory.ReservationStep;
387             SetPointers(_memory);
388             header = ref GetHeaderReference();
389             header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
390             ↪ LinkSizeInBytes);
391         }
392         header.AllocatedLinks = Increment(header.AllocatedLinks);
393         _memory.UsedCapacity += LinkSizeInBytes;
394         freeLink = header.AllocatedLinks;
395     }
396     return freeLink;
397 }
398
399 [MethodImpl(MethodImplOptions.AggressiveInlining)]
400 public virtual void Delete(ICollection<TLink> restrictions)
401 {
402     ref var header = ref GetHeaderReference();
403     var link = restrictions[Constants.IndexPart];
404     if (LessThan(link, header.AllocatedLinks)
405     {
406         UnusedLinksListMethods.AttachAsFirst(link);
407     }
408     else if (AreEqual(link, header.AllocatedLinks))
409     {
410         header.AllocatedLinks = Decrement(header.AllocatedLinks);
411         _memory.UsedCapacity -= LinkSizeInBytes;
412         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
413         ↪ пока не дойдём до первой существующей связи
414         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
415         while (GreaterThan(header.AllocatedLinks, GetZero()) &&
416         ↪ IsUnusedLink(header.AllocatedLinks))
417         {
418             UnusedLinksListMethods.Detach(header.AllocatedLinks);
419             header.AllocatedLinks = Decrement(header.AllocatedLinks);
420             _memory.UsedCapacity -= LinkSizeInBytes;
421         }
422     }
423 }
424
425 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

422 public IList<TLink> GetLinkStruct(TLink linkIndex)
423 {
424     ref var link = ref GetLinkReference(linkIndex);
425     return new Link<TLink>(linkIndex, link.Source, link.Target);
426 }
427
428 /// <remarks>
429 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
430   ↪ адрес реально поменялся
431 ///
432 /// Указатель this.links может быть в том же месте,
433 /// так как 0-я связь не используется и имеет такой же размер как Header,
434 /// поэтому header размещается в том же месте, что и 0-я связь
435 /// </remarks>
436 [MethodImpl(MethodImplOptions.AggressiveInlining)]
437 protected abstract void SetPointers(IResizableDirectMemory memory);
438
439 [MethodImpl(MethodImplOptions.AggressiveInlining)]
440 protected virtual void ResetPointers()
441 {
442     SourcesTreeMethods = null;
443     TargetsTreeMethods = null;
444     UnusedLinksListMethods = null;
445 }
446
447 [MethodImpl(MethodImplOptions.AggressiveInlining)]
448 protected abstract ref LinksHeader<TLink> GetHeaderReference();
449
450 [MethodImpl(MethodImplOptions.AggressiveInlining)]
451 protected abstract ref RawLink<TLink> GetLinkReference(TLink linkIndex);
452
453 [MethodImpl(MethodImplOptions.AggressiveInlining)]
454 protected virtual bool Exists(TLink link)
455     => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
456     && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
457     && !IsUnusedLink(link);
458
459 [MethodImpl(MethodImplOptions.AggressiveInlining)]
460 protected virtual bool IsUnusedLink(TLink linkIndex)
461 {
462     if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
463         ↪ is not needed
464     {
465         ref var link = ref GetLinkReference(linkIndex);
466         return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
467     }
468     else
469     {
470         return true;
471     }
472 }
473
474 [MethodImpl(MethodImplOptions.AggressiveInlining)]
475 protected virtual TLink GetOne() => _one;
476
477 [MethodImpl(MethodImplOptions.AggressiveInlining)]
478 protected virtual TLink GetZero() => default;
479
480 [MethodImpl(MethodImplOptions.AggressiveInlining)]
481 protected virtual bool AreEqual(TLink first, TLink second) =>
482     ↪ _equalityComparer.Equals(first, second);
483
484 [MethodImpl(MethodImplOptions.AggressiveInlining)]
485 protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
486     ↪ second) < 0;
487
488 [MethodImpl(MethodImplOptions.AggressiveInlining)]
489 protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
490     ↪ _comparer.Compare(first, second) <= 0;
491
492 [MethodImpl(MethodImplOptions.AggressiveInlining)]
493 protected virtual bool GreaterThan(TLink first, TLink second) =>
494     ↪ _comparer.Compare(first, second) > 0;
495
496 [MethodImpl(MethodImplOptions.AggressiveInlining)]
497 protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
498     ↪ _comparer.Compare(first, second) >= 0;
499
500 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

494     protected virtual long ConvertToInt64(TLink value) =>
495         ↪ _addressToInt64Converter.Convert(value);
496
497     [MethodImpl(MethodImplOptions.AggressiveInlining)]
498     protected virtual TLink ConvertToAddress(long value) =>
499         ↪ _int64ToAddressConverter.Convert(value);
500
501     [MethodImpl(MethodImplOptions.AggressiveInlining)]
502     protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
503         ↪ second);
504
505     [MethodImpl(MethodImplOptions.AggressiveInlining)]
506     protected virtual TLink Subtract(TLink first, TLink second) =>
507         ↪ Arithmetic<TLink>.Subtract(first, second);
508
509     [MethodImpl(MethodImplOptions.AggressiveInlining)]
510     protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
511
512     [MethodImpl(MethodImplOptions.AggressiveInlining)]
513     protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
514
515     #region Disposable
516
517     protected override bool AllowMultipleDisposeCalls
518     {
519         [MethodImpl(MethodImplOptions.AggressiveInlining)]
520         get => true;
521     }
522
523     [MethodImpl(MethodImplOptions.AggressiveInlining)]
524     protected override void Dispose(bool manual, bool wasDisposed)
525     {
526         if (!wasDisposed)
527         {
528             ResetPointers();
529             _memory.DisposeIfPossible();
530         }
531     }
532
533     #endregion
534 }

```

1.51 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/UnusedLinksListMethods.cs

```

1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Methods.Lists;
3  using Platform.Converters;
4  using static System.Runtime.CompilerServices.Unsafe;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
9  {
10     public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
11         ↪ ILinksListMethods<TLink>
12     {
13         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
14             ↪ UncheckedConverter<TLink, long>.Default;
15
16         private readonly byte* _links;
17         private readonly byte* _header;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnusedLinksListMethods(byte* links, byte* header)
21         {
22             _links = links;
23             _header = header;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
28             ↪ AsRef<LinksHeader<TLink>>(_header);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
32             ↪ AsRef<RawLink<TLink>>(_links + (RawLink<TLink>.SizeInBytes *
33             ↪ _addressToInt64Converter.Convert(link)));
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;

```

```

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 }

```

1.52 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/ILinksListMethods.cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.ResizableDirectMemory
6  {
7      public interface ILinksListMethods<TLink>
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         void Detach(TLink freeLink);
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         void AttachAsFirst(TLink link);
14     }
15 }

```

1.53 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/ILinksTreeMethods.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.ResizableDirectMemory
8  {
9      public interface ILinksTreeMethods<TLink>
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         TLink CountUsages(TLink link);
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         TLink Search(TLink source, TLink target);
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         TLink EachUsage(TLink source, Func<IList<TLink>, TLink> handler);
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         void Detach(ref TLink firstAsSource, TLink linkIndex);
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         void Attach(ref TLink firstAsSource, TLink linkIndex);
25     }
26 }

```


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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Unsafe;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.ResizableDirectMemory
9  {
10     public struct LinksHeader<TLink> : IEquatable<LinksHeader<TLink>>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↳ EqualityComparer<TLink>.Default;
14
15         public static readonly long SizeInBytes = Structure<LinksHeader<TLink>>.Size;
16
17         public TLink AllocatedLinks;
18         public TLink ReservedLinks;
19         public TLink FreeLinks;
20         public TLink FirstFreeLink;
21         public TLink FirstAsSource;
22         public TLink FirstAsTarget;
23         public TLink LastFreeLink;
24         public TLink Reserved8;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public override bool Equals(object obj) => obj is LinksHeader<TLink> linksHeader ?
28             ↳ Equals(linksHeader) : false;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public bool Equals(LinksHeader<TLink> other)
32             => _equalityComparer.Equals(AllocatedLinks, other.AllocatedLinks)
33             && _equalityComparer.Equals(ReservedLinks, other.ReservedLinks)
34             && _equalityComparer.Equals(FreeLinks, other.FreeLinks)
35             && _equalityComparer.Equals(FirstFreeLink, other.FirstFreeLink)
36             && _equalityComparer.Equals(FirstAsSource, other.FirstAsSource)
37             && _equalityComparer.Equals(FirstAsTarget, other.FirstAsTarget)
38             && _equalityComparer.Equals(LastFreeLink, other.LastFreeLink)
39             && _equalityComparer.Equals(Reserved8, other.Reserved8);
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public override int GetHashCode() => (AllocatedLinks, ReservedLinks, FreeLinks,
43             ↳ FirstFreeLink, FirstAsSource, FirstAsTarget, LastFreeLink, Reserved8).GetHashCode();
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public static bool operator ==(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
47             ↳ left.Equals(right);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public static bool operator !=(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
51             ↳ !(left == right);
52     }
53 }

```

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

```

1  using Platform.Unsafe;
2  using System;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.ResizableDirectMemory
9  {
10     public struct RawLink<TLink> : IEquatable<RawLink<TLink>>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↳ EqualityComparer<TLink>.Default;
14
15         public static readonly long SizeInBytes = Structure<RawLink<TLink>>.Size;
16
17         public TLink Source;
18         public TLink Target;
19         public TLink LeftAsSource;
20         public TLink RightAsSource;
21         public TLink SizeAsSource;
22         public TLink LeftAsTarget;
23         public TLink RightAsTarget;
24         public TLink SizeAsTarget;
25     }
26 }

```

```

25 [MethodImpl(MethodImplOptions.AggressiveInlining)]
26 public override bool Equals(object obj) => obj is RawLink<TLink> link ? Equals(link) :
    ↳ false;
27
28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 public bool Equals(RawLink<TLink> other)
30     => _equalityComparer.Equals(Source, other.Source)
31     && _equalityComparer.Equals(Target, other.Target)
32     && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
33     && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
34     && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
35     && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
36     && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
37     && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 public override int GetHashCode() => (Source, Target, LeftAsSource, RightAsSource,
    ↳ SizeAsSource, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 public static bool operator ==(RawLink<TLink> left, RawLink<TLink> right) =>
    ↳ left.Equals(right);
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 public static bool operator !=(RawLink<TLink> left, RawLink<TLink> right) => !(left ==
    ↳ right);
47 }
48 }

```

1.56 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksAvlBalancedTreeMethodsBase

```

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 :
    ↳ LinksAvlBalancedTreeMethodsBase<ulong>
10     {
11         protected new readonly RawLink<ulong>* Links;
12         protected new readonly LinksHeader<ulong>* Header;
13
14         protected UInt64LinksAvlBalancedTreeMethodsBase(LinksConstants<ulong> constants,
    ↳ RawLink<ulong>* links, LinksHeader<ulong>* header)
15             : base(constants, (byte*)links, (byte*)header)
16         {
17             Links = links;
18             Header = header;
19         }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override ulong GetZero() => 0UL;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override bool EqualToZero(ulong value) => value == 0UL;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override bool AreEqual(ulong first, ulong second) => first == second;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override bool GreaterThanZero(ulong value) => value > 0UL;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override bool GreaterThan(ulong first, ulong second) => first > second;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
    ↳ always true for ulong
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
    ↳ always >= 0 for ulong
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
    ↪ for ulong

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool LessThan(ulong first, ulong second) => first < second;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override ulong Increment(ulong value) => ++value;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override ulong Decrement(ulong value) => --value;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override ulong Add(ulong first, ulong second) => first + second;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override ulong Subtract(ulong first, ulong second) => first - second;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
{
    ref var firstLink = ref Links[first];
    ref var secondLink = ref Links[second];
    return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
        ↪ secondLink.Source, secondLink.Target);
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
{
    ref var firstLink = ref Links[first];
    ref var secondLink = ref Links[second];
    return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
        ↪ secondLink.Source, secondLink.Target);
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override ulong GetSizeValue(ulong value) => (value & 4294967264UL) >> 5;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
    ↪ storedValue & 31UL | (size & 134217727UL) << 5;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool GetLeftIsChildValue(ulong value) => (value & 16UL) >> 4 == 1UL;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
    ↪ storedValue = storedValue & 4294967279UL | (As<bool, byte>(ref value) & 1UL) << 4;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool GetRightIsChildValue(ulong value) => (value & 8UL) >> 3 == 1UL;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>
    ↪ storedValue = storedValue & 4294967287UL | (As<bool, byte>(ref value) & 1UL) << 3;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override sbyte GetBalanceValue(ulong value) => unchecked((sbyte)(value & 7UL |
    ↪ 0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of
    ↪ sbyte

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override void SetBalanceValue(ref ulong storedValue, sbyte value) =>
    ↪ storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
    ↪ value & 3) & 7UL);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];

```

```

}

```

1.57 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSizeBalancedTreeMethodsBase

```
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         protected UInt64LinksSizeBalancedTreeMethodsBase(LinksConstants<ulong> constants,
15             ↳ RawLink<ulong>* links, LinksHeader<ulong>* header)
16             : base(constants, (byte*)links, (byte*)header)
17         {
18             Links = links;
19             Header = header;
20
21             [MethodImpl(MethodImplOptions.AggressiveInlining)]
22             protected override ulong GetZero() => 0UL;
23
24             [MethodImpl(MethodImplOptions.AggressiveInlining)]
25             protected override bool EqualToZero(ulong value) => value == 0UL;
26
27             [MethodImpl(MethodImplOptions.AggressiveInlining)]
28             protected override bool AreEqual(ulong first, ulong second) => first == second;
29
30             [MethodImpl(MethodImplOptions.AggressiveInlining)]
31             protected override bool GreaterThanZero(ulong value) => value > 0UL;
32
33             [MethodImpl(MethodImplOptions.AggressiveInlining)]
34             protected override bool GreaterThan(ulong first, ulong second) => first > second;
35
36             [MethodImpl(MethodImplOptions.AggressiveInlining)]
37             protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
38
39             [MethodImpl(MethodImplOptions.AggressiveInlining)]
40             protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
41             ↳ always true for ulong
42
43             [MethodImpl(MethodImplOptions.AggressiveInlining)]
44             protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
45             ↳ always >= 0 for ulong
46
47             [MethodImpl(MethodImplOptions.AggressiveInlining)]
48             protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
49
50             [MethodImpl(MethodImplOptions.AggressiveInlining)]
51             protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
52             ↳ for ulong
53
54             [MethodImpl(MethodImplOptions.AggressiveInlining)]
55             protected override bool LessThan(ulong first, ulong second) => first < second;
56
57             [MethodImpl(MethodImplOptions.AggressiveInlining)]
58             protected override ulong Increment(ulong value) => ++value;
59
60             [MethodImpl(MethodImplOptions.AggressiveInlining)]
61             protected override ulong Decrement(ulong value) => --value;
62
63             [MethodImpl(MethodImplOptions.AggressiveInlining)]
64             protected override ulong Add(ulong first, ulong second) => first + second;
65
66             [MethodImpl(MethodImplOptions.AggressiveInlining)]
67             protected override ulong Subtract(ulong first, ulong second) => first - second;
68
69             [MethodImpl(MethodImplOptions.AggressiveInlining)]
70             protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
71             {
72                 ref var firstLink = ref Links[first];
73                 ref var secondLink = ref Links[second];
74                 return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
75                     ↳ secondLink.Source, secondLink.Target);
76             }
77
78             [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```

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

```

1.58 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesAvlBalancedTreeMeth

```

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
11         public UInt64LinksSourcesAvlBalancedTreeMethods(LinksConstants<ulong> constants,
12             ↪ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
13             ↪ { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected override ref ulong GetLeftReference(ulong node) => ref
17             ↪ Links[node].LeftAsSource;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected override ref ulong GetRightReference(ulong node) => ref
21             ↪ Links[node].RightAsSource;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
31             ↪ left;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
35             ↪ right;
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsSource);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
42             ↪ Links[node].SizeAsSource, size);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override bool GetLeftIsChild(ulong node) =>
46             ↪ GetLeftIsChildValue(Links[node].SizeAsSource);
47
48         // [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         // protected override bool GetLeftIsChild(ulong node) => IsChild(node, GetLeft(node));
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override void SetLeftIsChild(ulong node, bool value) =>
53             ↪ SetLeftIsChildValue(ref Links[node].SizeAsSource, value);
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override bool GetRightIsChild(ulong node) =>
57             ↪ GetRightIsChildValue(Links[node].SizeAsSource);
58
59         // [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         // protected override bool GetRightIsChild(ulong node) => IsChild(node, GetRight(node));
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

51     protected override void SetRightIsChild(ulong node, bool value) =>
52         ↳ SetRightIsChildValue(ref Links[node].SizeAsSource, value);
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected override sbyte GetBalance(ulong node) =>
56         ↳ GetBalanceValue(Links[node].SizeAsSource);
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
60         ↳ Links[node].SizeAsSource, value);
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected override ulong GetTreeRoot() => Header->FirstAsSource;
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
70         ↳ ulong secondSource, ulong secondTarget)
71         => firstSource < secondSource || (firstSource == secondSource && firstTarget <
72             ↳ secondTarget);
73
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
76         ↳ ulong secondSource, ulong secondTarget)
77         => firstSource > secondSource || (firstSource == secondSource && firstTarget >
78             ↳ secondTarget);
79
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     protected override void ClearNode(ulong node)
82     {
83         ref var link = ref Links[node];
84         link.LeftAsSource = OUL;
85         link.RightAsSource = OUL;
86         link.SizeAsSource = OUL;
87     }
88 }

```

1.59 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesSizeBalancedTreeMet

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
6  {
7      public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
8          ↳ UInt64LinksSizeBalancedTreeMethodsBase
9      {
10         public UInt64LinksSourcesSizeBalancedTreeMethods(LinksConstants<ulong> constants,
11             ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
12             ↳ { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetLeftReference(ulong node) => ref
16             ↳ Links[node].LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20             ↳ Links[node].RightAsSource;
21
22         [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) => Links[node].SizeAsSource;
38     }
39 }

```

```

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

```

1.60 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsAvlBalancedTreeMeth

```

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 :
    ↪ UInt64LinksAvlBalancedTreeMethodsBase
8     {
9         public UInt64LinksTargetsAvlBalancedTreeMethods(LinksConstants<ulong> constants,
    ↪ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
    ↪ { }
10
11     [MethodImpl(MethodImplOptions.AggressiveInlining)]
12     protected override ref ulong GetLeftReference(ulong node) => ref
    ↪ Links[node].LeftAsTarget;
13
14     [MethodImpl(MethodImplOptions.AggressiveInlining)]
15     protected override ref ulong GetRightReference(ulong node) => ref
    ↪ Links[node].RightAsTarget;
16
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
    ↪ left;
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
    ↪ right;
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsTarget);
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
    ↪ Links[node].SizeAsTarget, size);
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override bool GetLeftIsChild(ulong node) =>
    ↪ GetLeftIsChildValue(Links[node].SizeAsTarget);

```

```

37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override void SetLeftIsChild(ulong node, bool value) =>
39         ↳ SetLeftIsChildValue(ref Links[node].SizeAsTarget, value);
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override bool GetRightIsChild(ulong node) =>
43         ↳ GetRightIsChildValue(Links[node].SizeAsTarget);
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override void SetRightIsChild(ulong node, bool value) =>
47         ↳ SetRightIsChildValue(ref Links[node].SizeAsTarget, value);
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override sbyte GetBalance(ulong node) =>
51         ↳ GetBalanceValue(Links[node].SizeAsTarget);
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,
61         ↳ ulong secondSource, ulong secondTarget)
62         => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
63             ↳ secondSource);
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
67         ↳ ulong secondSource, ulong secondTarget)
68         => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
69             ↳ secondSource);
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     protected override void ClearNode(ulong node)
73     {
74         ref var link = ref Links[node];
75         link.LeftAsTarget = OUL;
76         link.RightAsTarget = OUL;
77         link.SizeAsTarget = OUL;
78     }
79 }

```

1.61 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsSizeBalancedTreeMet

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
6 {
7     public unsafe class UInt64LinksTargetsSizeBalancedTreeMethods :
8         ↳ UInt64LinksSizeBalancedTreeMethodsBase
9     {
10         public UInt64LinksTargetsSizeBalancedTreeMethods(LinksConstants<ulong> constants,
11             ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
12             ↳ { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetLeftReference(ulong node) => ref
16             ↳ Links[node].LeftAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20             ↳ Links[node].RightAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
27
28     }
29 }

```



```

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

```

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

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Memory;
4  using Platform.Data.Doublets.ResizableDirectMemory.Generic;
5  using Platform.Singletons;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
10 {
11     /// <summary>
12     /// <para>Represents a low-level implementation of direct access to resizable memory, for
    ↪ organizing the storage of links with addresses represented as <see cref="System.UInt64"
    ↪ </para>
13     /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
    ↪ размером, для организации хранения связей с адресами представленными в виде <see
    ↪ cref="System.UInt64"> </para>
14     /// </summary>
15     public unsafe class UInt64ResizableDirectMemoryLinks : ResizableDirectMemoryLinksBase<ulong>
16     {
17         private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
18         private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
19         private LinksHeader<ulong>* _header;
20         private RawLink<ulong>* _links;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public UInt64ResizableDirectMemoryLinks(string address) : this(address,
    ↪ DefaultLinksSizeStep) { }
24
25         /// <summary>
26         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
    ↪ 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
    ↪ байтах.</param>
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
    ↪ this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
    ↪ memoryReservationStep) { }

32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
    ↪ DefaultLinksSizeStep) { }

35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
    ↪ memoryReservationStep) : this(memory, memoryReservationStep,
    ↪ Default<LinksConstants<ulong>>.Instance, true) { }

38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
    ↪ memoryReservationStep, LinksConstants<ulong> constants, bool useAvlBasedIndex) :
    ↪ base(memory, memoryReservationStep, constants)
41     {
42         if (useAvlBasedIndex)
43         {
44             _createSourceTreeMethods = () => new
    ↪ UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
45             _createTargetTreeMethods = () => new
    ↪ UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
46         }
47         else
48         {
49             _createSourceTreeMethods = () => new
    ↪ UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
50             _createTargetTreeMethods = () => new
    ↪ UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
51         }
52         Init(memory, memoryReservationStep);
53     }

54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override void SetPointers(IResizableDirectMemory memory)
57     {
58         _header = (LinksHeader<ulong>*)memory.Pointer;
59         _links = (RawLink<ulong>*)memory.Pointer;
60         SourcesTreeMethods = _createSourceTreeMethods();
61         TargetsTreeMethods = _createTargetTreeMethods();
62         UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
63     }

64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override void ResetPointers()
67     {
68         base.ResetPointers();
69         _links = null;
70         _header = null;
71     }

72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;

75
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
    ↪ _links[linkIndex];

78
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     protected override bool AreEqual(ulong first, ulong second) => first == second;

81
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     protected override bool LessThan(ulong first, ulong second) => first < second;

84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;

87
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     protected override bool GreaterThan(ulong first, ulong second) => first > second;

90
91     [MethodImpl(MethodImplOptions.AggressiveInlining)]
92     protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;

93

```

```

94     [MethodImpl(MethodImplOptions.AggressiveInlining)]
95     protected override ulong GetZero() => 0UL;
96
97     [MethodImpl(MethodImplOptions.AggressiveInlining)]
98     protected override ulong GetOne() => 1UL;
99
100    [MethodImpl(MethodImplOptions.AggressiveInlining)]
101    protected override long ConvertToInt64(ulong value) => (long)value;
102
103    [MethodImpl(MethodImplOptions.AggressiveInlining)]
104    protected override ulong ConvertToAddress(long value) => (ulong)value;
105
106    [MethodImpl(MethodImplOptions.AggressiveInlining)]
107    protected override ulong Add(ulong first, ulong second) => first + second;
108
109    [MethodImpl(MethodImplOptions.AggressiveInlining)]
110    protected override ulong Subtract(ulong first, ulong second) => first - second;
111
112    [MethodImpl(MethodImplOptions.AggressiveInlining)]
113    protected override ulong Increment(ulong link) => ++link;
114
115    [MethodImpl(MethodImplOptions.AggressiveInlining)]
116    protected override ulong Decrement(ulong link) => --link;
117 }
118 }

```

1.63 ./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64UnusedLinksListMethods.cs

```

1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.ResizableDirectMemory.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
7  {
8      public unsafe class UInt64UnusedLinksListMethods : UnusedLinksListMethods<ulong>
9      {
10         private readonly RawLink<ulong>* _links;
11         private readonly LinksHeader<ulong>* _header;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public UInt64UnusedLinksListMethods(RawLink<ulong>* links, LinksHeader<ulong>* header)
15             : base((byte*)links, (byte*)header)
16         {
17             _links = links;
18             _header = header;
19         }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref _links[link];
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
26     }
27 }

```

1.64 ./csharp/Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Converters
7  {
8      public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public override TLink Convert(ICollection<TLink> sequence)
15         {
16             var length = sequence.Count;
17             if (length < 1)
18             {
19                 return default;
20             }
21             if (length == 1)
22             {
23                 return sequence[0];
24             }
25         }
26     }
27 }

```

```

25 // Make copy of next layer
26 if (length > 2)
27 {
28     // TODO: Try to use stackalloc (which at the moment is not working with
29     // ↪ generics) but will be possible with Sigil
30     var halvedSequence = new TLink[(length / 2) + (length % 2)];
31     HalveSequence(halvedSequence, sequence, length);
32     sequence = halvedSequence;
33     length = halvedSequence.Length;
34 }
35 // Keep creating layer after layer
36 while (length > 2)
37 {
38     HalveSequence(sequence, sequence, length);
39     length = (length / 2) + (length % 2);
40 }
41 return _links.GetOrCreate(sequence[0], sequence[1]);
42 }
43 [MethodImpl(MethodImplOptions.AggressiveInlining)]
44 private void HalveSequence(ICollection<TLink> destination, ICollection<TLink> source, int length)
45 {
46     var loopedLength = length - (length % 2);
47     for (var i = 0; i < loopedLength; i += 2)
48     {
49         destination[i / 2] = _links.GetOrCreate(source[i], source[i + 1]);
50     }
51     if (length > loopedLength)
52     {
53         destination[length / 2] = source[length - 1];
54     }
55 }
56 }
57 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Collections;
5 using Platform.Converters;
6 using Platform.Singletons;
7 using Platform.Numbers;
8 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.Sequences.Converters
13 {
14     /// <remarks>
15     /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от
16     /// ↪ Links на этапе сжатия.
17     /// А именно будет создаваться временный список пар необходимых для выполнения сжатия, в
18     /// ↪ таком случае тип значения элемента массива может быть любым, как char так и ulong.
19     /// Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
20     /// ↪ пар, а так же разом выполнить замену.
21     /// </remarks>
22     public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
23     {
24         private static readonly LinksConstants<TLink> _constants =
25             ↪ Default<LinksConstants<TLink>>.Instance;
26         private static readonly EqualityComparer<TLink> _equalityComparer =
27             ↪ EqualityComparer<TLink>.Default;
28         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
29
30         private static readonly TLink _zero = default;
31         private static readonly TLink _one = Arithmetic.Increment(_zero);
32
33         private readonly IConverter<ICollection<TLink>, TLink> _baseConverter;
34         private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
35         private readonly TLink _minFrequencyToCompress;
36         private readonly bool _doInitialFrequenciesIncrement;
37         private Doublet<TLink> _maxDoublet;
38         private LinkFrequency<TLink> _maxDoubletData;
39
40         private struct HalfDoublet
41         {
42             public TLink Element;
43             public LinkFrequency<TLink> DoubletData;
44         }
45     }
46 }

```

```

40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
42     {
43         Element = element;
44         DoubletData = doubletData;
45     }
46
47     public override string ToString() => $"{Element}: ({DoubletData})";
48 }
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
    : this(links, baseConverter, doubletFrequenciesCache, _one, true) { }
52
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
    ↪ doInitialFrequenciesIncrement)
    : this(links, baseConverter, doubletFrequenciesCache, _one,
    ↪ doInitialFrequenciesIncrement) { }
56
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
    ↪ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
    : base(links)
60 {
61     _baseConverter = baseConverter;
62     _doubletFrequenciesCache = doubletFrequenciesCache;
63     if (_comparer.Compare(minFrequencyToCompress, _one) < 0)
64     {
65         minFrequencyToCompress = _one;
66     }
67     _minFrequencyToCompress = minFrequencyToCompress;
68     _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
69     ResetMaxDoublet();
70 }
71
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 public override TLink Convert(IList<TLink> source) =>
    ↪ _baseConverter.Convert(Compress(source));
75
76
77 /// <remarks>
78 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
79 /// Faster version (doublets' frequencies dictionary is not recreated).
80 /// </remarks>
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 private IList<TLink> Compress(IList<TLink> sequence)
83 {
84     if (sequence.IsNullOrEmpty())
85     {
86         return null;
87     }
88     if (sequence.Count == 1)
89     {
90         return sequence;
91     }
92     if (sequence.Count == 2)
93     {
94         return new[] { _links.GetOrCreate(sequence[0], sequence[1]) };
95     }
96     // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
97     var copy = new HalfDoublet[sequence.Count];
98     Doublet<TLink> doublet = default;
99     for (var i = 1; i < sequence.Count; i++)
100     {
101         doublet.Source = sequence[i - 1];
102         doublet.Target = sequence[i];
103         LinkFrequency<TLink> data;
104         if (_doInitialFrequenciesIncrement)
105         {
106             data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
107         }
108         else
109         {
110             data = _doubletFrequenciesCache.GetFrequency(ref doublet);
111             if (data == null)
112             {

```

```

112         throw new NotSupportedException("If you ask not to increment
        ↪ frequencies, it is expected that all frequencies for the sequence
        ↪ are prepared.");
113     }
114 }
115 copy[i - 1].Element = sequence[i - 1];
116 copy[i - 1].DoubletData = data;
117 UpdateMaxDoublet(ref doublet, data);
118 }
119 copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
120 copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
121 if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
122 {
123     var newLength = ReplaceDoublets(copy);
124     sequence = new TLink[newLength];
125     for (int i = 0; i < newLength; i++)
126     {
127         sequence[i] = copy[i].Element;
128     }
129 }
130 return sequence;
131 }
132
133 /// <remarks>
134 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
135 /// </remarks>
136 [MethodImpl(MethodImplOptions.AggressiveInlining)]
137 private int ReplaceDoublets(HalfDoublet[] copy)
138 {
139     var oldLength = copy.Length;
140     var newLength = copy.Length;
141     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
142     {
143         var maxDoubletSource = _maxDoublet.Source;
144         var maxDoubletTarget = _maxDoublet.Target;
145         if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
146         {
147             _maxDoubletData.Link = _links.GetOrCreate(maxDoubletSource,
        ↪ maxDoubletTarget);
148         }
149         var maxDoubletReplacementLink = _maxDoubletData.Link;
150         oldLength--;
151         var oldLengthMinusTwo = oldLength - 1;
152         // Substitute all usages
153         int w = 0, r = 0; // (r == read, w == write)
154         for (; r < oldLength; r++)
155         {
156             if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
        ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
157             {
158                 if (r > 0)
159                 {
160                     var previous = copy[w - 1].Element;
161                     copy[w - 1].DoubletData.DecrementFrequency();
162                     copy[w - 1].DoubletData =
        ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
        ↪ maxDoubletReplacementLink);
163                 }
164                 if (r < oldLengthMinusTwo)
165                 {
166                     var next = copy[r + 2].Element;
167                     copy[r + 1].DoubletData.DecrementFrequency();
168                     copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(max
        ↪ xDoubletReplacementLink,
        ↪ next);
169                 }
170                 copy[w++].Element = maxDoubletReplacementLink;
171                 r++;
172                 newLength--;
173             }
174             else
175             {
176                 copy[w++] = copy[r];
177             }
178         }
179         if (w < newLength)
180         {
181             copy[w] = copy[r];

```

```

182     }
183     oldLength = newLength;
184     ResetMaxDoublet();
185     UpdateMaxDoublet(copy, newLength);
186 }
187 return newLength;
188 }
189
190 [MethodImpl(MethodImplOptions.AggressiveInlining)]
191 private void ResetMaxDoublet()
192 {
193     _maxDoublet = new Doublet<TLink>();
194     _maxDoubletData = new LinkFrequency<TLink>();
195 }
196
197 [MethodImpl(MethodImplOptions.AggressiveInlining)]
198 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
199 {
200     Doublet<TLink> doublet = default;
201     for (var i = 1; i < length; i++)
202     {
203         doublet.Source = copy[i - 1].Element;
204         doublet.Target = copy[i].Element;
205         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
206     }
207 }
208
209 [MethodImpl(MethodImplOptions.AggressiveInlining)]
210 private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
211 {
212     var frequency = data.Frequency;
213     var maxFrequency = _maxDoubletData.Frequency;
214     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
215     ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
216     ↪ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
217     ↪ _maxDoublet.Target)))
218     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
219     ↪ (_comparer.Compare(maxFrequency, frequency) < 0 ||
220     ↪ (_equalityComparer.Equals(maxFrequency, frequency) &&
221     ↪ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
222     ↪ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
223     ↪ better stability and better compression on sequent data and even on runderom
224     ↪ numbers data (but gives collisions anyway) */
225     {
226         _maxDoublet = doublet;
227         _maxDoubletData = data;
228     }
229 }
230 }
231 }
232 }
233 }

```

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

```

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

```

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

```

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

```

```

8 namespace Platform.Data.Doublets.Sequences.Converters
9 {
10     public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<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         private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
20             ↳ sequenceToItsLocalElementLevelsConverter) : base(links)
21             => _sequenceToItsLocalElementLevelsConverter =
22                 ↳ sequenceToItsLocalElementLevelsConverter;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public override TLink Convert(IList<TLink> sequence)
26         {
27             var length = sequence.Count;
28             if (length == 1)
29             {
30                 return sequence[0];
31             }
32             if (length == 2)
33             {
34                 return _links.GetOrCreate(sequence[0], sequence[1]);
35             }
36             sequence = sequence.ToArray();
37             var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
38             while (length > 2)
39             {
40                 var levelRepeat = 1;
41                 var currentLevel = levels[0];
42                 var previousLevel = levels[0];
43                 var skipOnce = false;
44                 var w = 0;
45                 for (var i = 1; i < length; i++)
46                 {
47                     if (_equalityComparer.Equals(currentLevel, levels[i]))
48                     {
49                         levelRepeat++;
50                         skipOnce = false;
51                         if (levelRepeat == 2)
52                         {
53                             sequence[w] = _links.GetOrCreate(sequence[i - 1], sequence[i]);
54                             var newLevel = i >= length - 1 ?
55                                 GetPreviousLowerThanCurrentOrCurrent(previousLevel,
56                                     ↳ currentLevel) :
57                                 i < 2 ?
58                                     GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
59                                     GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
60                                         ↳ currentLevel, levels[i + 1]);
61                             levels[w] = newLevel;
62                             previousLevel = currentLevel;
63                             w++;
64                             levelRepeat = 0;
65                             skipOnce = true;
66                         }
67                     }
68                     else if (i == length - 1)
69                     {
70                         sequence[w] = sequence[i];
71                         levels[w] = levels[i];
72                         w++;
73                     }
74                 }
75                 else
76                 {
77                     currentLevel = levels[i];
78                     levelRepeat = 1;
79                     if (skipOnce)
80                     {
81                         skipOnce = false;
82                     }
83                     else
84                     {
85                         sequence[w] = sequence[i - 1];
86                         levels[w] = levels[i - 1];
87                         previousLevel = levels[w];
88                         w++;
89                     }
90                 }
91             }
92         }
93     }
94 }

```



```

83         }
84         if (i == length - 1)
85         {
86             sequence[w] = sequence[i];
87             levels[w] = levels[i];
88             w++;
89         }
90     }
91     }
92     length = w;
93 }
94 return _links.GetOrCreate(sequence[0], sequence[1]);
95 }
96
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 private static TLink GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink previous, TLink
↪ current, TLink next)
99 {
100     return _comparer.Compare(previous, next) > 0
101         ? _comparer.Compare(previous, current) < 0 ? previous : current
102         : _comparer.Compare(next, current) < 0 ? next : current;
103 }
104
105 [MethodImpl(MethodImplOptions.AggressiveInlining)]
106 private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
↪ _comparer.Compare(next, current) < 0 ? next : current;
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
↪ => _comparer.Compare(previous, current) < 0 ? previous : current;
110 }
111 }

```

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

```

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

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Interfaces;

```

```

3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.CriterionMatchers
7 {
8     public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
9         ↳ ICriterionMatcher<TLink>
10    {
11        [MethodImpl(MethodImplOptions.AggressiveInlining)]
12        public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
13
14        [MethodImpl(MethodImplOptions.AggressiveInlining)]
15        public bool IsMatched(TLink argument) => _links.IsPartialPoint(argument);
16    }
17 }

```

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

```

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

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Collections.Stacks;
4 using Platform.Data.Doublets.Sequences.HeightProviders;
5 using Platform.Data.Sequences;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Sequences
10 {
11     public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
12         ↳ ISequenceAppender<TLink>
13    {
14        private static readonly EqualityComparer<TLink> _equalityComparer =
15            ↳ EqualityComparer<TLink>.Default;
16
17        private readonly IStack<TLink> _stack;
18        private readonly ISequenceHeightProvider<TLink> _heightProvider;
19
20        [MethodImpl(MethodImplOptions.AggressiveInlining)]
21        public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
22            ↳ ISequenceHeightProvider<TLink> heightProvider)
23            : base(links)
24        {
25            _stack = stack;
26            _heightProvider = heightProvider;
27        }
28
29        [MethodImpl(MethodImplOptions.AggressiveInlining)]
30        public TLink Append(TLink sequence, TLink appendant)
31        {
32
33        }
34    }
35 }

```

```

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

```

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

```

1  using System.Collections.Generic;
2  using System.Linq;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences
9  {
10     public class DuplicateSegmentsCounter<TLink> : ICounter<int>
11     {
12         private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
13             ↪ _duplicateFragmentsProvider;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
17             ↪ IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
18             ↪ duplicateFragmentsProvider;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
22     }
23 }

```

1.73 ./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Interfaces;
6  using Platform.Collections;
7  using Platform.Collections.Lists;
8  using Platform.Collections.Segments;
9  using Platform.Collections.Segments.Walkers;
10 using Platform.Singletons;
11 using Platform.Converters;
12 using Platform.Data.Doublets.Unicode;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     public class DuplicateSegmentsProvider<TLink> :
19         ↪ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
20         ↪ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
21     {
22         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
23             ↪ UncheckedConverter<TLink, long>.Default;
24         private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
25             ↪ UncheckedConverter<TLink, ulong>.Default;
26     }
27 }

```

```

22 private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
    ↳ UncheckedConverter<ulong, TLink>.Default;
23
24 private readonly ILinks<TLink> _links;
25 private readonly ILinks<TLink> _sequences;
26 private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
27 private BitString _visited;
28
29 private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
    ↳ IList<TLink>>>
30 {
31     private readonly IListEqualityComparer<TLink> _listComparer;
32
33     public ItemEquilityComparer() => _listComparer =
    ↳ Default<IListEqualityComparer<TLink>>.Instance;
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     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);
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
    ↳ (_listComparer.GetHashCode(pair.Key),
    ↳ _listComparer.GetHashCode(pair.Value)).GetHashCode();
40 }
41
42 private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
43 {
44     private readonly IListComparer<TLink> _listComparer;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
    ↳ KeyValuePair<IList<TLink>, IList<TLink>> right)
51     {
52         var intermediateResult = _listComparer.Compare(left.Key, right.Key);
53         if (intermediateResult == 0)
54         {
55             intermediateResult = _listComparer.Compare(left.Value, right.Value);
56         }
57         return intermediateResult;
58     }
59 }
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
63     : base(minimumStringSegmentLength: 2)
64 {
65     _links = links;
66     _sequences = sequences;
67 }
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
71 {
72     _groups = new HashSet<KeyValuePair<IList<TLink>,
    ↳ IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
73     var links = _links;
74     var count = links.Count();
75     _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
76     links.Each(link =>
77     {
78         var linkIndex = links.GetIndex(link);
79         var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
80         var constants = links.Constants;
81         if (!_visited.Get(linkBitIndex))
82         {
83             var sequenceElements = new List<TLink>();
84             var filler = new ListFiller<TLink, TLink>(sequenceElements, constants.Break);
85             _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
    ↳ LinkAddress<TLink>(linkIndex));
86             if (sequenceElements.Count > 2)
87             {
88                 WalkAll(sequenceElements);
89             }

```

```

90     }
91     return constants.Continue;
92 });
93 var resultList = _groups.ToList();
94 var comparer = Default<ItemComparer>.Instance;
95 resultList.Sort(comparer);
96 #if DEBUG
97     foreach (var item in resultList)
98     {
99         PrintDuplicates(item);
100     }
101 #endif
102     return resultList;
103 }
104
105 [MethodImpl(MethodImplOptions.AggressiveInlining)]
106 protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
    ↪ length) => new Segment<TLink>(elements, offset, length);
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 protected override void OnDuplicateFound(Segment<TLink> segment)
110 {
111     var duplicates = CollectDuplicatesForSegment(segment);
112     if (duplicates.Count > 1)
113     {
114         _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
            ↪ duplicates));
115     }
116 }
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
120 {
121     var duplicates = new List<TLink>();
122     var readAsElement = new HashSet<TLink>();
123     var restrictions = segment.ShiftRight();
124     var constants = _links.Constants;
125     restrictions[0] = constants.Any;
126     _sequences.Each(sequence =>
127     {
128         var sequenceIndex = sequence[constants.IndexPart];
129         duplicates.Add(sequenceIndex);
130         readAsElement.Add(sequenceIndex);
131         return constants.Continue;
132     }, restrictions);
133     if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
134     {
135         return new List<TLink>();
136     }
137     foreach (var duplicate in duplicates)
138     {
139         var duplicateBitIndex = _addressToInt64Converter.Convert(duplicate);
140         _visited.Set(duplicateBitIndex);
141     }
142     if (_sequences is Sequences sequencesExperiments)
143     {
144         var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
            ↪ ashSet<ulong>)(object)readAsElement,
            ↪ (IList<ulong>)segment);
145         foreach (var partiallyMatchedSequence in partiallyMatched)
146         {
147             var sequenceIndex =
            ↪ _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
148             duplicates.Add(sequenceIndex);
149         }
150     }
151     duplicates.Sort();
152     return duplicates;
153 }
154
155 [MethodImpl(MethodImplOptions.AggressiveInlining)]
156 private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
157 {
158     if (!(_links is ILinks<ulong> ulongLinks))
159     {
160         return;
161     }
162     var duplicatesKey = duplicatesItem.Key;

```

```

163     var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
164     Console.WriteLine($"{> {keyString} ({string.Join(", ", duplicatesKey)})");
165     var duplicatesList = duplicatesItem.Value;
166     for (int i = 0; i < duplicatesList.Count; i++)
167     {
168         var sequenceIndex = _addressToUInt64Converter.Convert(duplicatesList[i]);
169         var formattedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
            ↳ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
            ↳ UnicodeMap.IsCharLink(link.Index) ?
            ↳ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
170         Console.WriteLine(formattedSequenceStructure);
171         var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
            ↳ ulongLinks);
172         Console.WriteLine(sequenceString);
173     }
174     Console.WriteLine();
175 }
176 }
177 }

```

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

```

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

```

```

56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
58 {
59     var doublet = new Doublet<TLink>(source, target);
60     return IncrementFrequency(ref doublet);
61 }
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 public void PrintFrequencies(IList<TLink> sequence)
65 {
66     for (var i = 1; i < sequence.Count; i++)
67     {
68         PrintFrequency(sequence[i - 1], sequence[i]);
69     }
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 public void PrintFrequency(TLink source, TLink target)
74 {
75     var number = GetFrequency(source, target).Frequency;
76     Console.WriteLine("{0},{1} - {2}", source, target, number);
77 }
78
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
81 {
82     if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
83     {
84         data.IncrementFrequency();
85     }
86     else
87     {
88         var link = _links.SearchOrDefault(doublet.Source, doublet.Target);
89         data = new LinkFrequency<TLink>(_one, link);
90         if (!_equalityComparer.Equals(link, default))
91         {
92             data.Frequency = Arithmetic.Add(data.Frequency,
93                 ↪ _frequencyCounter.Count(link));
94         }
95         _doubletsCache.Add(doublet, data);
96     }
97     return data;
98 }
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 public void ValidateFrequencies()
102 {
103     foreach (var entry in _doubletsCache)
104     {
105         var value = entry.Value;
106         var linkIndex = value.Link;
107         if (!_equalityComparer.Equals(linkIndex, default))
108         {
109             var frequency = value.Frequency;
110             var count = _frequencyCounter.Count(linkIndex);
111             // TODO0: Why `frequency` always greater than `count` by 1?
112             if (((_comparer.Compare(frequency, count) > 0) &&
113                 ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count), _one) > 0))
114                 || ((_comparer.Compare(count, frequency) > 0) &&
115                 ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency), _one) > 0)))
116             {
117                 throw new InvalidOperationException("Frequencies validation failed.");
118             }
119             //else
120             //{
121             //    if (value.Frequency > 0)
122             //    {
123             //        var frequency = value.Frequency;
124             //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
125             //        var count = _countLinkFrequency(linkIndex);
126             //        if ((frequency > count && frequency - count > 1) || (count > frequency
127             //            ↪ && count - frequency > 1))
128             //            throw new InvalidOperationException("Frequencies validation
129             //            ↪ failed.");
130             //    }
131             //}

```

```

129         //}
130     }
131 }
132 }
133 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Numbers;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
7 {
8     public class LinkFrequency<TLink>
9     {
10         public TLink Frequency { get; set; }
11         public TLink Link { get; set; }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public LinkFrequency(TLink frequency, TLink link)
15         {
16             Frequency = frequency;
17             Link = link;
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public LinkFrequency() { }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public void IncrementFrequency() => Frequency = Arithmetic<TLink>.Increment(Frequency);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public override string ToString() => $"F: {Frequency}, L: {Link}";
31     }
32 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Converters;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
7 {
8     public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
9         ⇨ IConverter<Doublet<TLink>, TLink>
10     {
11         private readonly LinkFrequenciesCache<TLink> _cache;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public
15         ⇨ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
16         ⇨ cache) => _cache = cache;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
20     }
21 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Interfaces;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7 {
8     public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
9         ⇨ SequenceSymbolFrequencyOneOffCounter<TLink>
10     {
11         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
15         ⇨ ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
16     {
17     }
18 }

```



```

14         : base(links, sequenceLink, symbol)
15         => _markedSequenceMatcher = markedSequenceMatcher;
16
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     public override TLink Count()
19     {
20         if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
21         {
22             return default;
23         }
24         return base.Count();
25     }
26 }
27 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4  using Platform.Numbers;
5  using Platform.Data.Sequences;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
10 {
11     public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↪ EqualityComparer<TLink>.Default;
15         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
16
17         protected readonly ILinks<TLink> _links;
18         protected readonly TLink _sequenceLink;
19         protected readonly TLink _symbol;
20         protected TLink _total;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
24             ↪ TLink symbol)
25         {
26             _links = links;
27             _sequenceLink = sequenceLink;
28             _symbol = symbol;
29             _total = default;
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public virtual TLink Count()
34         {
35             if (_comparer.Compare(_total, default) > 0)
36             {
37                 return _total;
38             }
39             StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
40                 ↪ IsElement, VisitElement);
41             return _total;
42         }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
46             ↪ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
47             ↪ IsPartialPoint
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         private bool VisitElement(TLink element)
51         {
52             if (_equalityComparer.Equals(element, _symbol))
53             {
54                 _total = Arithmetic.Increment(_total);
55             }
56             return true;
57         }
58     }
59 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Interfaces;
3

```

```

4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7  {
8      public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
9      {
10         private readonly ILinks<TLink> _links;
11         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
15             ↪ ICriterionMatcher<TLink> markedSequenceMatcher)
16         {
17             _links = links;
18             _markedSequenceMatcher = markedSequenceMatcher;
19         }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public TLink Count(TLink argument) => new
23             ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
24             ↪ _markedSequenceMatcher, argument).Count();
25     }
26 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Interfaces;
3  using Platform.Numbers;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
8  {
9      public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
10         ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>
11     {
12         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
16             ↪ ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
17             : base(links, symbol)
18             => _markedSequenceMatcher = markedSequenceMatcher;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override void CountSequenceSymbolFrequency(TLink link)
22         {
23             var symbolFrequencyCounter = new
24                 ↪ MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
25                 ↪ _markedSequenceMatcher, link, _symbol);
26             _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
27         }
28     }
29 }

```

1.81 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter

```

1  using System.Runtime.CompilerServices;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7  {
8      public class TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
9      {
10         private readonly ILinks<TLink> _links;
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public TLink Count(TLink symbol) => new
17             ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
18     }
19 }

```

1.82 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter

```

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

```

```

3 using Platform.Interfaces;
4 using Platform.Numbers;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
9 {
10     public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↪ EqualityComparer<TLink>.Default;
14         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
15
16         protected readonly ILinks<TLink> _links;
17         protected readonly TLink _symbol;
18         protected readonly HashSet<TLink> _visits;
19         protected TLink _total;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
23         {
24             _links = links;
25             _symbol = symbol;
26             _visits = new HashSet<TLink>();
27             _total = default;
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public TLink Count()
32         {
33             if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
34             {
35                 return _total;
36             }
37             CountCore(_symbol);
38             return _total;
39         }
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         private void CountCore(TLink link)
43         {
44             var any = _links.Constants.Any;
45             if (_equalityComparer.Equals(_links.Count(any, link), default))
46             {
47                 CountSequenceSymbolFrequency(link);
48             }
49             else
50             {
51                 _links.Each(EachElementHandler, any, link);
52             }
53         }
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected virtual void CountSequenceSymbolFrequency(TLink link)
57         {
58             var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
59                 ↪ link, _symbol);
60             _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
61         }
62
63         [MethodImpl(MethodImplOptions.AggressiveInlining)]
64         private TLink EachElementHandler(IList<TLink> doublet)
65         {
66             var constants = _links.Constants;
67             var doubletIndex = doublet[constants.IndexPart];
68             if (_visits.Add(doubletIndex))
69             {
70                 CountCore(doubletIndex);
71             }
72             return constants.Continue;
73         }
74     }
75 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Interfaces;
4 using Platform.Converters;
5

```

```

6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.HeightProviders
9  {
10     public class CachedSequenceHeightProvider<TLink> : ISequenceHeightProvider<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↳ EqualityComparer<TLink>.Default;
14
15         private readonly TLink _heightPropertyMarker;
16         private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
17         private readonly IConverter<TLink> _addressToUnaryNumberConverter;
18         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
19         private readonly IProperties<TLink, TLink, TLink> _propertyOperator;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public CachedSequenceHeightProvider(
23             ISequenceHeightProvider<TLink> baseHeightProvider,
24             IConverter<TLink> addressToUnaryNumberConverter,
25             IConverter<TLink> unaryNumberToAddressConverter,
26             TLink heightPropertyMarker,
27             IProperties<TLink, TLink, TLink> propertyOperator)
28         {
29             _heightPropertyMarker = heightPropertyMarker;
30             _baseHeightProvider = baseHeightProvider;
31             _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
32             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
33             _propertyOperator = propertyOperator;
34         }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public TLink Get(TLink sequence)
38         {
39             TLink height;
40             var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
41             if (_equalityComparer.Equals(heightValue, default))
42             {
43                 height = _baseHeightProvider.Get(sequence);
44                 heightValue = _addressToUnaryNumberConverter.Convert(height);
45                 _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
46             }
47             else
48             {
49                 height = _unaryNumberToAddressConverter.Convert(heightValue);
50             }
51             return height;
52         }
53     }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Interfaces;
3  using Platform.Numbers;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences.HeightProviders
8  {
9     public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
10         ↳ ISequenceHeightProvider<TLink>
11     {
12         private readonly ICriterionMatcher<TLink> _elementMatcher;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
16             ↳ elementMatcher) : base(links) => _elementMatcher = elementMatcher;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public TLink Get(TLink sequence)
20         {
21             var height = default(TLink);
22             var pairOrElement = sequence;
23             while (!_elementMatcher.IsMatched(pairOrElement))
24             {
25                 pairOrElement = _links.GetTarget(pairOrElement);
26                 height = Arithmetic.Increment(height);
27             }
28             return height;
29         }
30     }

```

```

28     }
29 }

```

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

```

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

```

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

```

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

```

```

62         return false;
63     }
64     return !_equalityComparer.Equals(frequency.Frequency, default);
65 }
66 }
67 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4  using Platform.Incrementers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.Indexes
9  {
10     public class FrequencyIncrementingSequenceIndex<TLink> : SequenceIndex<TLink>,
11         ↳ ISequenceIndex<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↳ EqualityComparer<TLink>.Default;
15
16         private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
17         private readonly IIncrementer<TLink> _frequencyIncrementer;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IProperty<TLink, TLink>
21             ↳ frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
22             : base(links)
23         {
24             _frequencyPropertyOperator = frequencyPropertyOperator;
25             _frequencyIncrementer = frequencyIncrementer;
26         }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public override bool Add(IList<TLink> sequence)
30         {
31             var indexed = true;
32             var i = sequence.Count;
33             while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
34                 ↳ { }
35             for (; i >= 1; i--)
36             {
37                 Increment(_links.GetOrCreate(sequence[i - 1], sequence[i]));
38             }
39             return indexed;
40         }
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         private bool IsIndexedWithIncrement(TLink source, TLink target)
44         {
45             var link = _links.SearchOrCreate(source, target);
46             var indexed = !_equalityComparer.Equals(link, default);
47             if (indexed)
48             {
49                 Increment(link);
50             }
51             return indexed;
52         }
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         private void Increment(TLink link)
56         {
57             var previousFrequency = _frequencyPropertyOperator.Get(link);
58             var frequency = _frequencyIncrementer.Increment(previousFrequency);
59             _frequencyPropertyOperator.Set(link, frequency);
60         }
61     }
62 }

```

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

```

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

```

```

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

```

1.89 ./csharp/Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Indexes
7  {
8      public class SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↪ EqualityComparer<TLink>.Default;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public SequenceIndex(ILinks<TLink> links) : base(links) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public virtual bool Add(IList<TLink> sequence)
18         {
19             var indexed = true;
20             var i = sequence.Count;
21             while (--i >= 1 && (indexed =
22                 ↪ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
23                 ↪ default))) { }
24             for (; i >= 1; i--)
25             {
26                 _links.GetOrCreate(sequence[i - 1], sequence[i]);
27             }
28             return indexed;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public virtual bool MightContain(IList<TLink> sequence)
33         {
34             var indexed = true;
35             var i = sequence.Count;
36             while (--i >= 1 && (indexed =
37                 ↪ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
38                 ↪ default))) { }
39             return indexed;
40         }
41     }
42 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Indexes
7  {
8      public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↪ EqualityComparer<TLink>.Default;
12
13         private readonly ISynchronizedLinks<TLink> _links;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public bool Add(IList<TLink> sequence)

```

```

19     {
20         var indexed = true;
21         var i = sequence.Count;
22         var links = _links.Unsync;
23         _links.SyncRoot.ExecuteReadOperation(() =>
24         {
25             while (--i >= 1 && (indexed =
26                 ↪ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
27                 ↪ sequence[i]), default))) { }
28         });
29         if (!indexed)
30         {
31             _links.SyncRoot.ExecuteWriteOperation(() =>
32             {
33                 for (; i >= 1; i--)
34                 {
35                     links.GetOrCreate(sequence[i - 1], sequence[i]);
36                 }
37             });
38             return indexed;
39         }
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public bool MightContain(ICollection<TLink> sequence)
42         {
43             var links = _links.Unsync;
44             return _links.SyncRoot.ExecuteReadOperation(() =>
45             {
46                 var indexed = true;
47                 var i = sequence.Count;
48                 while (--i >= 1 && (indexed =
49                     ↪ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
50                     ↪ sequence[i]), default))) { }
51                 return indexed;
52             });
53         }
54     }
55 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Indexes
7 {
8     public class Unindex<TLink> : ISequenceIndex<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public virtual bool Add(ICollection<TLink> sequence) => false;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public virtual bool MightContain(ICollection<TLink> sequence) => true;
15     }
16 }

```

1.92 ./csharp/Platform.Data.Doublets/Sequences/Sequences.Experiments.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using System.Linq;
5 using System.Text;
6 using Platform.Collections;
7 using Platform.Collections.Sets;
8 using Platform.Collections.Stacks;
9 using Platform.Data.Exceptions;
10 using Platform.Data.Sequences;
11 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
12 using Platform.Data.Doublets.Sequences.Walkers;
13 using LinkIndex = System.UInt64;
14 using Stack = System.Collections.Generic.Stack<ulong>;
15
16 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
17
18 namespace Platform.Data.Doublets.Sequences
19 {
20     partial class Sequences
21     {

```



```

22 #region Create All Variants (Not Practical)
23
24 /// <remarks>
25 /// Number of links that is needed to generate all variants for
26 /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
27 /// </remarks>
28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 public ulong[] CreateAllVariants2(ulong[] sequence)
30 {
31     return _sync.ExecuteWriteOperation(() =>
32     {
33         if (sequence.IsNullOrEmpty())
34         {
35             return Array.Empty<ulong>();
36         }
37         Links.EnsureLinkExists(sequence);
38         if (sequence.Length == 1)
39         {
40             return sequence;
41         }
42         return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
43     });
44 }
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
48 {
49     #if DEBUG
50         if ((stopAt - startAt) < 0)
51         {
52             throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
53                 ↳ меньше или равен stopAt");
54         }
55         #endif
56         if ((stopAt - startAt) == 0)
57         {
58             return new[] { sequence[startAt] };
59         }
60         if ((stopAt - startAt) == 1)
61         {
62             return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt]) };
63         }
64         var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
65         var last = 0;
66         for (var splitter = startAt; splitter < stopAt; splitter++)
67         {
68             var left = CreateAllVariants2Core(sequence, startAt, splitter);
69             var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
70             for (var i = 0; i < left.Length; i++)
71             {
72                 for (var j = 0; j < right.Length; j++)
73                 {
74                     var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
75                     if (variant == Constants.Null)
76                     {
77                         throw new NotImplementedException("Creation cancellation is not
78                             ↳ implemented.");
79                     }
80                     variants[last++] = variant;
81                 }
82             }
83         }
84         return variants;
85     }
86 }
87
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 public List<ulong> CreateAllVariants1(params ulong[] sequence)
90 {
91     return _sync.ExecuteWriteOperation(() =>
92     {
93         if (sequence.IsNullOrEmpty())
94         {
95             return new List<ulong>();
96         }
97         Links.Unsync.EnsureLinkExists(sequence);
98         if (sequence.Length == 1)
99         {
100             return new List<ulong> { sequence[0] };
101         }
102     });
103 }

```

```

98     }
99     var results = new
    ↪ List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
100     return CreateAllVariants1Core(sequence, results);
101 });
102 }
103
104 [MethodImpl(MethodImplOptions.AggressiveInlining)]
105 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
106 {
107     if (sequence.Length == 2)
108     {
109         var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
110         if (link == Constants.Null)
111         {
112             throw new NotImplementedException("Creation cancellation is not
    ↪ implemented.");
113         }
114         results.Add(link);
115         return results;
116     }
117     var innerSequenceLength = sequence.Length - 1;
118     var innerSequence = new ulong[innerSequenceLength];
119     for (var li = 0; li < innerSequenceLength; li++)
120     {
121         var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
122         if (link == Constants.Null)
123         {
124             throw new NotImplementedException("Creation cancellation is not
    ↪ implemented.");
125         }
126         for (var isi = 0; isi < li; isi++)
127         {
128             innerSequence[isi] = sequence[isi];
129         }
130         innerSequence[li] = link;
131         for (var isi = li + 1; isi < innerSequenceLength; isi++)
132         {
133             innerSequence[isi] = sequence[isi + 1];
134         }
135         CreateAllVariants1Core(innerSequence, results);
136     }
137     return results;
138 }
139
140 #endregion
141
142 [MethodImpl(MethodImplOptions.AggressiveInlining)]
143 public HashSet<ulong> Each1(params ulong[] sequence)
144 {
145     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
146     Each1(link =>
147     {
148         if (!visitedLinks.Contains(link))
149         {
150             visitedLinks.Add(link); // изучить почему случаются повторы
151         }
152         return true;
153     }, sequence);
154     return visitedLinks;
155 }
156
157 [MethodImpl(MethodImplOptions.AggressiveInlining)]
158 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
159 {
160     if (sequence.Length == 2)
161     {
162         Links.Unsync.Each(sequence[0], sequence[1], handler);
163     }
164     else
165     {
166         var innerSequenceLength = sequence.Length - 1;
167         for (var li = 0; li < innerSequenceLength; li++)
168         {
169             var left = sequence[li];
170             var right = sequence[li + 1];
171             if (left == 0 && right == 0)
172             {

```

```

173         continue;
174     }
175     var linkIndex = li;
176     ulong[] innerSequence = null;
177     Links.Unsync.Each(doublet =>
178     {
179         if (innerSequence == null)
180         {
181             innerSequence = new ulong[innerSequenceLength];
182             for (var isi = 0; isi < linkIndex; isi++)
183             {
184                 innerSequence[isi] = sequence[isi];
185             }
186             for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
187             {
188                 innerSequence[isi] = sequence[isi + 1];
189             }
190         }
191         innerSequence[linkIndex] = doublet[Constants.IndexPart];
192         Each1(handler, innerSequence);
193         return Constants.Continue;
194     }, Constants.Any, left, right);
195     }
196 }
197
198 [MethodImpl(MethodImplOptions.AggressiveInlining)]
199 public HashSet<ulong> EachPart(params ulong[] sequence)
200 {
201     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
202     EachPartCore(link =>
203     {
204         var linkIndex = link[Constants.IndexPart];
205         if (!visitedLinks.Contains(linkIndex))
206         {
207             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
208         }
209         return Constants.Continue;
210     }, sequence);
211     return visitedLinks;
212 }
213
214 [MethodImpl(MethodImplOptions.AggressiveInlining)]
215 public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
216 {
217     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
218     EachPartCore(link =>
219     {
220         var linkIndex = link[Constants.IndexPart];
221         if (!visitedLinks.Contains(linkIndex))
222         {
223             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
224             return handler(new LinkAddress<LinkIndex>(linkIndex));
225         }
226         return Constants.Continue;
227     }, sequence);
228 }
229
230 [MethodImpl(MethodImplOptions.AggressiveInlining)]
231 private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
232 ↪ sequence)
233 {
234     if (sequence.IsNullOrEmpty())
235     {
236         return;
237     }
238     Links.EnsureLinkIsAnyOrExists(sequence);
239     if (sequence.Length == 1)
240     {
241         var link = sequence[0];
242         if (link > 0)
243         {
244             handler(new LinkAddress<LinkIndex>(link));
245         }
246         else
247         {
248             Links.Each(Constants.Any, Constants.Any, handler);
249         }
250     }
251 }

```

```

250     }
251     else if (sequence.Length == 2)
252     {
253         // _links.Each(sequence[0], sequence[1], handler);
254         //   o_|           x_o ...
255         // x_|           |__|
256         Links.Each(sequence[1], Constants.Any, doublet =>
257         {
258             var match = Links.SearchOrDefault(sequence[0], doublet);
259             if (match != Constants.Null)
260             {
261                 handler(new LinkAddress<LinkIndex>(match));
262             }
263             return true;
264         });
265         // |_x           ... x_o
266         // |_o           |__|
267         Links.Each(Constants.Any, sequence[0], doublet =>
268         {
269             var match = Links.SearchOrDefault(doublet, sequence[1]);
270             if (match != 0)
271             {
272                 handler(new LinkAddress<LinkIndex>(match));
273             }
274             return true;
275         });
276         //           . _x o _ .
277         //           |__|
278         PartialStepRight(x => handler(x), sequence[0], sequence[1]);
279     }
280     else
281     {
282         throw new NotImplementedException();
283     }
284 }
285
286 [MethodImpl(MethodImplOptions.AggressiveInlining)]
287 private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
288 {
289     Links.Unsync.Each(Constants.Any, left, doublet =>
290     {
291         StepRight(handler, doublet, right);
292         if (left != doublet)
293         {
294             PartialStepRight(handler, doublet, right);
295         }
296         return true;
297     });
298 }
299
300 [MethodImpl(MethodImplOptions.AggressiveInlining)]
301 private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
302 {
303     Links.Unsync.Each(left, Constants.Any, rightStep =>
304     {
305         TryStepRightUp(handler, right, rightStep);
306         return true;
307     });
308 }
309
310 [MethodImpl(MethodImplOptions.AggressiveInlining)]
311 private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
↪ stepFrom)
312 {
313     var upStep = stepFrom;
314     var firstSource = Links.Unsync.GetTarget(upStep);
315     while (firstSource != right && firstSource != upStep)
316     {
317         upStep = firstSource;
318         firstSource = Links.Unsync.GetSource(upStep);
319     }
320     if (firstSource == right)
321     {
322         handler(new LinkAddress<LinkIndex>(stepFrom));
323     }
324 }
325
326 // TODO: Test

```

```

327 [MethodImpl(MethodImplOptions.AggressiveInlining)]
328 private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
329 {
330     Links.Unsync.Each(right, Constants.Any, doublet =>
331     {
332         StepLeft(handler, left, doublet);
333         if (right != doublet)
334         {
335             PartialStepLeft(handler, left, doublet);
336         }
337         return true;
338     });
339 }
340
341 [MethodImpl(MethodImplOptions.AggressiveInlining)]
342 private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
343 {
344     Links.Unsync.Each(Constants.Any, right, leftStep =>
345     {
346         TryStepLeftUp(handler, left, leftStep);
347         return true;
348     });
349 }
350
351 [MethodImpl(MethodImplOptions.AggressiveInlining)]
352 private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
353 {
354     var upStep = stepFrom;
355     var firstTarget = Links.Unsync.GetSource(upStep);
356     while (firstTarget != left && firstTarget != upStep)
357     {
358         upStep = firstTarget;
359         firstTarget = Links.Unsync.GetTarget(upStep);
360     }
361     if (firstTarget == left)
362     {
363         handler(new LinkAddress<LinkIndex>(stepFrom));
364     }
365 }
366
367 [MethodImpl(MethodImplOptions.AggressiveInlining)]
368 private bool StartsWith(ulong sequence, ulong link)
369 {
370     var upStep = sequence;
371     var firstSource = Links.Unsync.GetSource(upStep);
372     while (firstSource != link && firstSource != upStep)
373     {
374         upStep = firstSource;
375         firstSource = Links.Unsync.GetSource(upStep);
376     }
377     return firstSource == link;
378 }
379
380 [MethodImpl(MethodImplOptions.AggressiveInlining)]
381 private bool EndsWith(ulong sequence, ulong link)
382 {
383     var upStep = sequence;
384     var lastTarget = Links.Unsync.GetTarget(upStep);
385     while (lastTarget != link && lastTarget != upStep)
386     {
387         upStep = lastTarget;
388         lastTarget = Links.Unsync.GetTarget(upStep);
389     }
390     return lastTarget == link;
391 }
392
393 [MethodImpl(MethodImplOptions.AggressiveInlining)]
394 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
395 {
396     return _sync.ExecuteReadOperation(() =>
397     {
398         var results = new List<ulong>();
399         if (sequence.Length > 0)
400         {
401             Links.EnsureLinkExists(sequence);
402             var firstElement = sequence[0];
403             if (sequence.Length == 1)
404             {
405                 results.Add(firstElement);

```

```

406         return results;
407     }
408     if (sequence.Length == 2)
409     {
410         var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
411         if (doublet != Constants.Null)
412         {
413             results.Add(doublet);
414         }
415         return results;
416     }
417     var linksInSequence = new HashSet<ulong>(sequence);
418     void handler(ICollection<LinkIndex> result)
419     {
420         var resultIndex = result[Links.Constants.IndexPart];
421         var filterPosition = 0;
422         StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
423             ↪ Links.Unsync.GetTarget,
424             ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
425             ↪ x =>
426             {
427                 if (filterPosition == sequence.Length)
428                 {
429                     filterPosition = -2; // Длиннее чем нужно
430                     return false;
431                 }
432                 if (x != sequence[filterPosition])
433                 {
434                     filterPosition = -1;
435                     return false; // Начинается иначе
436                 }
437                 filterPosition++;
438                 return true;
439             });
440         if (filterPosition == sequence.Length)
441         {
442             results.Add(resultIndex);
443         }
444     }
445     if (sequence.Length >= 2)
446     {
447         StepRight(handler, sequence[0], sequence[1]);
448     }
449     var last = sequence.Length - 2;
450     for (var i = 1; i < last; i++)
451     {
452         PartialStepRight(handler, sequence[i], sequence[i + 1]);
453     }
454     if (sequence.Length >= 3)
455     {
456         StepLeft(handler, sequence[sequence.Length - 2],
457             ↪ sequence[sequence.Length - 1]);
458     }
459     }
460     return results;
461 });
462
463 [MethodImpl(MethodImplOptions.AggressiveInlining)]
464 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
465 {
466     return _sync.ExecuteReadOperation(() =>
467     {
468         var results = new HashSet<ulong>();
469         if (sequence.Length > 0)
470         {
471             Links.EnsureLinkExists(sequence);
472             var firstElement = sequence[0];
473             if (sequence.Length == 1)
474             {
475                 results.Add(firstElement);
476                 return results;
477             }
478             if (sequence.Length == 2)
479             {
480                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
481                 if (doublet != Constants.Null)

```

```

481         {
482             results.Add(douplet);
483         }
484         return results;
485     }
486     var matcher = new Matcher(this, sequence, results, null);
487     if (sequence.Length >= 2)
488     {
489         StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
490     }
491     var last = sequence.Length - 2;
492     for (var i = 1; i < last; i++)
493     {
494         PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
495             ↪ sequence[i + 1]);
496     }
497     if (sequence.Length >= 3)
498     {
499         StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
500             ↪ sequence[sequence.Length - 1]);
501     }
502     return results;
503 });
504 }
505
506 public const int MaxSequenceFormatSize = 200;
507
508 [MethodImpl(MethodImplOptions.AggressiveInlining)]
509 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
510     ↪ => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
511
512 [MethodImpl(MethodImplOptions.AggressiveInlining)]
513 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
514     ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
515     ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
516     ↪ elementToString, insertComma, knownElements));
517
518 [MethodImpl(MethodImplOptions.AggressiveInlining)]
519 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
520     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
521     ↪ LinkIndex[] knownElements)
522 {
523     var linksInSequence = new HashSet<ulong>(knownElements);
524     //var entered = new HashSet<ulong>();
525     var sb = new StringBuilder();
526     sb.Append('{');
527     if (links.Exists(sequenceLink))
528     {
529         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
530             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
531             ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
532         {
533             if (insertComma && sb.Length > 1)
534             {
535                 sb.Append(',');
536             }
537             //if (entered.Contains(element))
538             //{
539             //    sb.Append('{');
540             //    elementToString(sb, element);
541             //    sb.Append('}');
542             //}
543             //else
544             elementToString(sb, element);
545             if (sb.Length < MaxSequenceFormatSize)
546             {
547                 return true;
548             }
549             sb.Append(insertComma ? ", ..." : "...");
550             return false;
551         }
552     }
553     sb.Append('}');
554     return sb.ToString();
555 }
556
557 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```



```

618         if (filterPosition >= 0)
619         {
620             if (x == sequence[filterPosition + 1])
621             {
622                 filterPosition++;
623             }
624             else
625             {
626                 return false;
627             }
628         }
629         if (filterPosition < 0)
630         {
631             if (x == sequence[0])
632             {
633                 filterPosition = 0;
634             }
635         }
636         return true;
637     });
638     if (filterPosition == (sequence.Length - 1))
639     {
640         filteredResults.Add(result);
641     }
642 }
643 return filteredResults;
644 }
645 return new List<ulong>();
646 });
647 }
648
649 [MethodImpl(MethodImplOptions.AggressiveInlining)]
650 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
651 {
652     return _sync.ExecuteReadOperation(() =>
653     {
654         if (sequence.Length > 0)
655         {
656             Links.EnsureLinkExists(sequence);
657             var results = new HashSet<ulong>();
658             for (var i = 0; i < sequence.Length; i++)
659             {
660                 AllUsagesCore(sequence[i], results);
661             }
662             var filteredResults = new HashSet<ulong>();
663             var matcher = new Matcher(this, sequence, filteredResults, null);
664             matcher.AddAllPartialMatchedToResults(results);
665             return filteredResults;
666         }
667         return new HashSet<ulong>();
668     });
669 }
670
671 [MethodImpl(MethodImplOptions.AggressiveInlining)]
672 public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
673     ↪ params ulong[] sequence)
674 {
675     return _sync.ExecuteReadOperation(() =>
676     {
677         if (sequence.Length > 0)
678         {
679             Links.EnsureLinkExists(sequence);
680
681             var results = new HashSet<ulong>();
682             var filteredResults = new HashSet<ulong>();
683             var matcher = new Matcher(this, sequence, filteredResults, handler);
684             for (var i = 0; i < sequence.Length; i++)
685             {
686                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
687                 {
688                     return false;
689                 }
690             }
691             return true;
692         }
693         return true;
694     });
695 }

```

```

696 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
697 //{
698 //    return Sync.ExecuteReadOperation(() =>
699 //    {
700 //        if (sequence.Length > 0)
701 //        {
702 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
703 //
704 //            var firstResults = new HashSet<ulong>();
705 //            var lastResults = new HashSet<ulong>();
706 //
707 //            var first = sequence.First(x => x != LinksConstants.Any);
708 //            var last = sequence.Last(x => x != LinksConstants.Any);
709 //
710 //            AllUsagesCore(first, firstResults);
711 //            AllUsagesCore(last, lastResults);
712 //
713 //            firstResults.IntersectWith(lastResults);
714 //
715 //            //for (var i = 0; i < sequence.Length; i++)
716 //            //    AllUsagesCore(sequence[i], results);
717 //
718 //            var filteredResults = new HashSet<ulong>();
719 //            var matcher = new Matcher(this, sequence, filteredResults, null);
720 //            matcher.AddAllPartialMatchedToResults(firstResults);
721 //            return filteredResults;
722 //        }
723 //
724 //        return new HashSet<ulong>();
725 //    });
726 //}
727
728 [MethodImpl(MethodImplOptions.AggressiveInlining)]
729 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
730 {
731     return _sync.ExecuteReadOperation((Func<HashSet<ulong>>)(() =>
732     {
733         if (sequence.Length > 0)
734         {
735             ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links,
736                 ↪ (IList<ulong>)sequence);
737             var firstResults = new HashSet<ulong>();
738             var lastResults = new HashSet<ulong>();
739             var first = sequence.First(x => x != Constants.Any);
740             var last = sequence.Last(x => x != Constants.Any);
741             AllUsagesCore(first, firstResults);
742             AllUsagesCore(last, lastResults);
743             firstResults.IntersectWith(lastResults);
744             //for (var i = 0; i < sequence.Length; i++)
745             //    AllUsagesCore(sequence[i], results);
746             var filteredResults = new HashSet<ulong>();
747             var matcher = new Matcher(this, sequence, filteredResults, null);
748             matcher.AddAllPartialMatchedToResults(firstResults);
749             return filteredResults;
750         }
751         return new HashSet<ulong>();
752     }));
753 }
754
755 [MethodImpl(MethodImplOptions.AggressiveInlining)]
756 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
757     ↪ IList<ulong> sequence)
758 {
759     return _sync.ExecuteReadOperation(() =>
760     {
761         if (sequence.Count > 0)
762         {
763             Links.EnsureLinkExists(sequence);
764             var results = new HashSet<LinkIndex>();
765             //var nextResults = new HashSet<ulong>();
766             //for (var i = 0; i < sequence.Length; i++)
767             //{
768             //    AllUsagesCore(sequence[i], nextResults);
769             //    if (results.IsNullOrEmpty())
770             //    {
771                 results = nextResults;
772                 nextResults = new HashSet<ulong>();
773             }
774         }
775     });
776 }

```

```

772         //     else
773         //     {
774         //         results.IntersectWith(nextResults);
775         //         nextResults.Clear();
776         //     }
777         //}
778         var collector1 = new AllUsagesCollector1(Links.Unsync, results);
779         collector1.Collect(Links.Unsync.GetLink(sequence[0]));
780         var next = new HashSet<ulong>();
781         for (var i = 1; i < sequence.Count; i++)
782         {
783             var collector = new AllUsagesCollector1(Links.Unsync, next);
784             collector.Collect(Links.Unsync.GetLink(sequence[i]));
785
786             results.IntersectWith(next);
787             next.Clear();
788         }
789         var filteredResults = new HashSet<ulong>();
790         var matcher = new Matcher(this, sequence, filteredResults, null,
791             ↪ readAsElements);
792         matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
793             ↪ x)); // OrderBy is a Hack
794         return filteredResults;
795     }
796     return new HashSet<ulong>();
797 });
798 }
799
800 // Does not work
801 //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
802 //    ↪ params ulong[] sequence)
803 //{
804 //    var visited = new HashSet<ulong>();
805 //    var results = new HashSet<ulong>();
806 //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
807 //    ↪ true; }, readAsElements);
808 //    var last = sequence.Length - 1;
809 //    for (var i = 0; i < last; i++)
810 //    {
811 //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
812 //    }
813 //    return results;
814 //}
815
816 [MethodImpl(MethodImplOptions.AggressiveInlining)]
817 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
818 {
819     return _sync.ExecuteReadOperation(() =>
820     {
821         if (sequence.Length > 0)
822         {
823             Links.EnsureLinkExists(sequence);
824             //var firstElement = sequence[0];
825             //if (sequence.Length == 1)
826             //{
827             //    //results.Add(firstElement);
828             //    return results;
829             //}
830             //if (sequence.Length == 2)
831             //{
832             //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
833             //    //if (doublet != Doublets.Links.Null)
834             //    //    results.Add(doublet);
835             //    return results;
836             //}
837             //var lastElement = sequence[sequence.Length - 1];
838             //Func<ulong, bool> handler = x =>
839             //{
840             //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
841             //    ↪ results.Add(x);
842             //    return true;
843             //};
844             //if (sequence.Length >= 2)
845             //    StepRight(handler, sequence[0], sequence[1]);
846             //var last = sequence.Length - 2;
847             //for (var i = 1; i < last; i++)
848             //    PartialStepRight(handler, sequence[i], sequence[i + 1]);

```

```

844 //if (sequence.Length >= 3)
845 //    StepLeft(handler, sequence[sequence.Length - 2],
846 //        sequence[sequence.Length - 1]);
847 //if (sequence.Length == 1)
848 //if (sequence.Length == 1)
849 //    throw new NotImplementedException(); // all sequences, containing
850 //    this element?
851 //if (sequence.Length == 2)
852 //if (sequence.Length == 2)
853 //    var results = new List<ulong>();
854 //    PartialStepRight(results.Add, sequence[0], sequence[1]);
855 //    return results;
856 //var matches = new List<List<ulong>>();
857 //var last = sequence.Length - 1;
858 //for (var i = 0; i < last; i++)
859 //if (sequence.Length == 2)
860 //    var results = new List<ulong>();
861 //    //StepRight(results.Add, sequence[i], sequence[i + 1]);
862 //    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
863 //    if (results.Count > 0)
864 //        matches.Add(results);
865 //    else
866 //        return results;
867 //    if (matches.Count == 2)
868 //    {
869 //        var merged = new List<ulong>();
870 //        for (var j = 0; j < matches[0].Count; j++)
871 //            for (var k = 0; k < matches[1].Count; k++)
872 //                CloseInnerConnections(merged.Add, matches[0][j],
873 //                    matches[1][k]);
874 //        if (merged.Count > 0)
875 //            matches = new List<List<ulong>> { merged };
876 //        else
877 //            return new List<ulong>();
878 //    }
879 //if (matches.Count > 0)
880 //if (matches.Count > 0)
881 //    var usages = new HashSet<ulong>();
882 //    for (int i = 0; i < sequence.Length; i++)
883 //    {
884 //        AllUsagesCore(sequence[i], usages);
885 //    }
886 //for (int i = 0; i < matches[0].Count; i++)
887 //    AllUsagesCore(matches[0][i], usages);
888 //usages.UnionWith(matches[0]);
889 //return usages.ToList();
890 //var firstLinkUsages = new HashSet<ulong>();
891 //AllUsagesCore(sequence[0], firstLinkUsages);
892 //firstLinkUsages.Add(sequence[0]);
893 //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
894 //    sequence[0] }; // or all sequences, containing this element?
895 //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
896 //    1).ToList();
897 //var results = new HashSet<ulong>();
898 //foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
899 //    firstLinkUsages, 1))
900 //    AllUsagesCore(match, results);
901 //return results.ToList();
902 }
903 return new List<ulong>();
904 });
905 }
906
907 <remarks>
908 TODO: Может потребоваться ограничение на уровень глубины рекурсии
909 </remarks>
910 [MethodImpl(MethodImplOptions.AggressiveInlining)]
911 public HashSet<ulong> AllUsages(ulong link)
912 {
913     return _sync.ExecuteReadOperation(() =>
914     {

```

```

915         var usages = new HashSet<ulong>();
916         AllUsagesCore(link, usages);
917         return usages;
918     });
919 }
920
921 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
922 //   ↳ той связи с которой начинался поиск (STTTSSSTT),
923 // причём достаточно одного бита для хранения перехода влево или вправо
924 [MethodImpl(MethodImplOptions.AggressiveInlining)]
925 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
926 {
927     bool handler(ulong doublet)
928     {
929         if (usages.Add(doublet))
930         {
931             AllUsagesCore(doublet, usages);
932         }
933         return true;
934     }
935     Links.Unsync.Each(link, Constants.Any, handler);
936     Links.Unsync.Each(Constants.Any, link, handler);
937 }
938
939 [MethodImpl(MethodImplOptions.AggressiveInlining)]
940 public HashSet<ulong> AllBottomUsages(ulong link)
941 {
942     return _sync.ExecuteReadOperation(() =>
943     {
944         var visits = new HashSet<ulong>();
945         var usages = new HashSet<ulong>();
946         AllBottomUsagesCore(link, visits, usages);
947         return usages;
948     });
949 }
950
951 [MethodImpl(MethodImplOptions.AggressiveInlining)]
952 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
953   ↳ usages)
954 {
955     bool handler(ulong doublet)
956     {
957         if (visits.Add(doublet))
958         {
959             AllBottomUsagesCore(doublet, visits, usages);
960         }
961         return true;
962     }
963     if (Links.Unsync.Count(Constants.Any, link) == 0)
964     {
965         usages.Add(link);
966     }
967     else
968     {
969         Links.Unsync.Each(link, Constants.Any, handler);
970         Links.Unsync.Each(Constants.Any, link, handler);
971     }
972 }
973
974 [MethodImpl(MethodImplOptions.AggressiveInlining)]
975 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
976 {
977     if (Options.UseSequenceMarker)
978     {
979         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
980   ↳ Options.MarkedSequenceMatcher, symbol);
981         return counter.Count();
982     }
983     else
984     {
985         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
986   ↳ symbol);
987         return counter.Count();
988     }
989 }
990
991 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

988 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
    ↳ LinkIndex> outerHandler)
989 {
990     bool handler(ulong doublet)
991     {
992         if (usages.Add(doublet))
993         {
994             if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
995             {
996                 return false;
997             }
998             if (!AllUsagesCore1(doublet, usages, outerHandler))
999             {
1000                 return false;
1001             }
1002         }
1003         return true;
1004     }
1005     return Links.Unsync.Each(link, Constants.Any, handler)
1006         && Links.Unsync.Each(Constants.Any, link, handler);
1007 }
1008
1009 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1010 public void CalculateAllUsages(ulong[] totals)
1011 {
1012     var calculator = new AllUsagesCalculator(Links, totals);
1013     calculator.Calculate();
1014 }
1015
1016 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1017 public void CalculateAllUsages2(ulong[] totals)
1018 {
1019     var calculator = new AllUsagesCalculator2(Links, totals);
1020     calculator.Calculate();
1021 }
1022
1023 private class AllUsagesCalculator
1024 {
1025     private readonly SynchronizedLinks<ulong> _links;
1026     private readonly ulong[] _totals;
1027
1028     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1029     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
1030     {
1031         _links = links;
1032         _totals = totals;
1033     }
1034
1035     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1036     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
    ↳ CalculateCore);
1037
1038     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1039     private bool CalculateCore(ulong link)
1040     {
1041         if (_totals[link] == 0)
1042         {
1043             var total = 1UL;
1044             _totals[link] = total;
1045             var visitedChildren = new HashSet<ulong>();
1046             bool linkCalculator(ulong child)
1047             {
1048                 if (link != child && visitedChildren.Add(child))
1049                 {
1050                     total += _totals[child] == 0 ? 1 : _totals[child];
1051                 }
1052                 return true;
1053             }
1054             _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1055             _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1056             _totals[link] = total;
1057         }
1058         return true;
1059     }
1060 }
1061
1062 private class AllUsagesCalculator2
1063 {
1064     private readonly SynchronizedLinks<ulong> _links;

```

```

1065 private readonly ulong[] _totals;
1066
1067 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1068 public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1069 {
1070     _links = links;
1071     _totals = totals;
1072 }
1073
1074 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1075 public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
    ↪ CalculateCore);
1076
1077 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1078 private bool IsElement(ulong link)
1079 {
1080     //_linksInSequence.Contains(link) ||
1081     return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
    ↪ link;
1082 }
1083
1084 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1085 private bool CalculateCore(ulong link)
1086 {
1087     // TODO: Проработать защиту от заикливания
1088     // Основано на SequenceWalker.WalkLeft
1089     Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1090     Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1091     Func<ulong, bool> isElement = IsElement;
1092     void visitLeaf(ulong parent)
1093     {
1094         if (link != parent)
1095         {
1096             _totals[parent]++;
1097         }
1098     }
1099     void visitNode(ulong parent)
1100     {
1101         if (link != parent)
1102         {
1103             _totals[parent]++;
1104         }
1105     }
1106     var stack = new Stack();
1107     var element = link;
1108     if (isElement(element))
1109     {
1110         visitLeaf(element);
1111     }
1112     else
1113     {
1114         while (true)
1115         {
1116             if (isElement(element))
1117             {
1118                 if (stack.Count == 0)
1119                 {
1120                     break;
1121                 }
1122                 element = stack.Pop();
1123                 var source = getSource(element);
1124                 var target = getTarget(element);
1125                 // Обработка элемента
1126                 if (isElement(target))
1127                 {
1128                     visitLeaf(target);
1129                 }
1130                 if (isElement(source))
1131                 {
1132                     visitLeaf(source);
1133                 }
1134                 element = source;
1135             }
1136             else
1137             {
1138                 stack.Push(element);
1139                 visitNode(element);
1140                 element = getTarget(element);
1141             }
1142         }
1143     }
1144 }

```

```

1142     }
1143 }
1144     _totals[link]++;
1145     return true;
1146 }
1147 }
1148
1149 private class AllUsagesCollector
1150 {
1151     private readonly ILinks<ulong> _links;
1152     private readonly HashSet<ulong> _usages;
1153
1154     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1155     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1156     {
1157         _links = links;
1158         _usages = usages;
1159     }
1160
1161     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1162     public bool Collect(ulong link)
1163     {
1164         if (_usages.Add(link))
1165         {
1166             _links.Each(link, _links.Constants.Any, Collect);
1167             _links.Each(_links.Constants.Any, link, Collect);
1168         }
1169         return true;
1170     }
1171 }
1172
1173 private class AllUsagesCollector1
1174 {
1175     private readonly ILinks<ulong> _links;
1176     private readonly HashSet<ulong> _usages;
1177     private readonly ulong _continue;
1178
1179     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1180     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1181     {
1182         _links = links;
1183         _usages = usages;
1184         _continue = _links.Constants.Continue;
1185     }
1186
1187     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1188     public ulong Collect(IList<ulong> link)
1189     {
1190         var linkIndex = _links.GetIndex(link);
1191         if (_usages.Add(linkIndex))
1192         {
1193             _links.Each(Collect, _links.Constants.Any, linkIndex);
1194         }
1195         return _continue;
1196     }
1197 }
1198
1199 private class AllUsagesCollector2
1200 {
1201     private readonly ILinks<ulong> _links;
1202     private readonly BitString _usages;
1203
1204     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1205     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1206     {
1207         _links = links;
1208         _usages = usages;
1209     }
1210
1211     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1212     public bool Collect(ulong link)
1213     {
1214         if (_usages.Add((long)link))
1215         {
1216             _links.Each(link, _links.Constants.Any, Collect);
1217             _links.Each(_links.Constants.Any, link, Collect);
1218         }
1219         return true;
1220     }
1221 }

```



```

1222 private class AllUsagesIntersectingCollector
1223 {
1224     private readonly SynchronizedLinks<ulong> _links;
1225     private readonly HashSet<ulong> _intersectWith;
1226     private readonly HashSet<ulong> _usages;
1227     private readonly HashSet<ulong> _enter;
1228
1229     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1230     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
1231     ↪ intersectWith, HashSet<ulong> usages)
1232     {
1233         _links = links;
1234         _intersectWith = intersectWith;
1235         _usages = usages;
1236         _enter = new HashSet<ulong>(); // защита от зацикливания
1237     }
1238
1239     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1240     public bool Collect(ulong link)
1241     {
1242         if (_enter.Add(link))
1243         {
1244             if (_intersectWith.Contains(link))
1245             {
1246                 _usages.Add(link);
1247             }
1248             _links.Unsync.Each(link, _links.Constants.Any, Collect);
1249             _links.Unsync.Each(_links.Constants.Any, link, Collect);
1250         }
1251         return true;
1252     }
1253 }
1254
1255 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1256 private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
1257 ↪ right)
1258 {
1259     TryStepLeftUp(handler, left, right);
1260     TryStepRightUp(handler, right, left);
1261 }
1262
1263 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1264 private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
1265 ↪ right)
1266 {
1267     // Direct
1268     if (left == right)
1269     {
1270         handler(new LinkAddress<LinkIndex>(left));
1271     }
1272     var doublet = Links.Unsync.SearchOrDefault(left, right);
1273     if (doublet != Constants.Null)
1274     {
1275         handler(new LinkAddress<LinkIndex>(doublet));
1276     }
1277     // Inner
1278     CloseInnerConnections(handler, left, right);
1279     // Outer
1280     StepLeft(handler, left, right);
1281     StepRight(handler, left, right);
1282     PartialStepRight(handler, left, right);
1283     PartialStepLeft(handler, left, right);
1284 }
1285
1286 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1287 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
1288 ↪ HashSet<ulong> previousMatchings, long startAt)
1289 {
1290     if (startAt >= sequence.Length) // ?
1291     {
1292         return previousMatchings;
1293     }
1294     var secondLinkUsages = new HashSet<ulong>();
1295     AllUsagesCore(sequence[startAt], secondLinkUsages);
1296     secondLinkUsages.Add(sequence[startAt]);
1297     var matchings = new HashSet<ulong>();
1298     var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);

```

```

1296 //for (var i = 0; i < previousMatchings.Count; i++)
1297 foreach (var secondLinkUsage in secondLinkUsages)
1298 {
1299     foreach (var previousMatching in previousMatchings)
1300     {
1301         //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
1302         ↪ secondLinkUsage);
1303         StepRight(filler.AddFirstAndReturnConstant, previousMatching,
1304         ↪ secondLinkUsage);
1305         TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
1306         ↪ previousMatching);
1307         //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1308         ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1309         ↪ желаемым результатам.
1310         PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
1311         ↪ secondLinkUsage);
1312     }
1313 }
1314 if (matchings.Count == 0)
1315 {
1316     return matchings;
1317 }
1318 return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1319 }
1320
1321 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1322 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
1323 ↪ links, params ulong[] sequence)
1324 {
1325     if (sequence == null)
1326     {
1327         return;
1328     }
1329     for (var i = 0; i < sequence.Length; i++)
1330     {
1331         if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
1332         ↪ !links.Exists(sequence[i]))
1333         {
1334             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1335             ↪ $"patternSequence[{i}]");
1336         }
1337     }
1338 }
1339 }
1340
1341 // Pattern Matching -> Key To Triggers
1342 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1343 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1344 {
1345     return _sync.ExecuteReadOperation(() =>
1346     {
1347         patternSequence = Simplify(patternSequence);
1348         if (patternSequence.Length > 0)
1349         {
1350             EnsureEachLinkIsAnyOrZeroOrManyOrExists(links, patternSequence);
1351             var uniqueSequenceElements = new HashSet<ulong>();
1352             for (var i = 0; i < patternSequence.Length; i++)
1353             {
1354                 if (patternSequence[i] != Constants.Any && patternSequence[i] !=
1355                 ↪ ZeroOrMany)
1356                 {
1357                     uniqueSequenceElements.Add(patternSequence[i]);
1358                 }
1359             }
1360             var results = new HashSet<ulong>();
1361             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1362             {
1363                 AllUsagesCore(uniqueSequenceElement, results);
1364             }
1365             var filteredResults = new HashSet<ulong>();
1366             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1367             matcher.AddAllPatternMatchedToResults(results);
1368             return filteredResults;
1369         }
1370         return new HashSet<ulong>();
1371     });
1372 }
1373 }

```

```

1363 // Найти все возможные связи между указанным списком связей.
1364 // Находит связи между всеми указанными связями в любом порядке.
1365 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
    ↳ несколько раз в последовательности)
1366 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1367 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1368 {
1369     return _sync.ExecuteReadOperation(() =>
1370     {
1371         var results = new HashSet<ulong>();
1372         if (linksToConnect.Length > 0)
1373         {
1374             Links.EnsureLinkExists(linksToConnect);
1375             AllUsagesCore(linksToConnect[0], results);
1376             for (var i = 1; i < linksToConnect.Length; i++)
1377             {
1378                 var next = new HashSet<ulong>();
1379                 AllUsagesCore(linksToConnect[i], next);
1380                 results.IntersectWith(next);
1381             }
1382         }
1383         return results;
1384     });
1385 }
1386
1387 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1388 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1389 {
1390     return _sync.ExecuteReadOperation(() =>
1391     {
1392         var results = new HashSet<ulong>();
1393         if (linksToConnect.Length > 0)
1394         {
1395             Links.EnsureLinkExists(linksToConnect);
1396             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1397             collector1.Collect(linksToConnect[0]);
1398             var next = new HashSet<ulong>();
1399             for (var i = 1; i < linksToConnect.Length; i++)
1400             {
1401                 var collector = new AllUsagesCollector(Links.Unsync, next);
1402                 collector.Collect(linksToConnect[i]);
1403                 results.IntersectWith(next);
1404                 next.Clear();
1405             }
1406         }
1407         return results;
1408     });
1409 }
1410
1411 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1412 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1413 {
1414     return _sync.ExecuteReadOperation(() =>
1415     {
1416         var results = new HashSet<ulong>();
1417         if (linksToConnect.Length > 0)
1418         {
1419             Links.EnsureLinkExists(linksToConnect);
1420             var collector1 = new AllUsagesCollector(Links, results);
1421             collector1.Collect(linksToConnect[0]);
1422             //AllUsagesCore(linksToConnect[0], results);
1423             for (var i = 1; i < linksToConnect.Length; i++)
1424             {
1425                 var next = new HashSet<ulong>();
1426                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1427                 collector.Collect(linksToConnect[i]);
1428                 //AllUsagesCore(linksToConnect[i], next);
1429                 //results.IntersectWith(next);
1430                 results = next;
1431             }
1432         }
1433         return results;
1434     });
1435 }
1436
1437 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1438 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1439 {

```

```

1440     return _sync.ExecuteReadOperation(()=>
1441     {
1442         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1443         ↪ BitArray((int)_links.Total + 1);
1444         if (linksToConnect.Length > 0)
1445         {
1446             Links.EnsureLinkExists(linksToConnect);
1447             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1448             collector1.Collect(linksToConnect[0]);
1449             for (var i = 1; i < linksToConnect.Length; i++)
1450             {
1451                 var next = new BitString((long)Links.Unsync.Count() + 1); //new
1452                 ↪ BitArray((int)_links.Total + 1);
1453                 var collector = new AllUsagesCollector2(Links.Unsync, next);
1454                 collector.Collect(linksToConnect[i]);
1455                 results = results.And(next);
1456             }
1457             return results.GetSetUInt64Indices();
1458         });
1459     }
1460     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1461     private static ulong[] Simplify(ulong[] sequence)
1462     {
1463         // Считаем новый размер последовательности
1464         long newLength = 0;
1465         var zeroOrManyStepped = false;
1466         for (var i = 0; i < sequence.Length; i++)
1467         {
1468             if (sequence[i] == ZeroOrMany)
1469             {
1470                 if (zeroOrManyStepped)
1471                 {
1472                     continue;
1473                 }
1474                 zeroOrManyStepped = true;
1475             }
1476             else
1477             {
1478                 //if (zeroOrManyStepped) Is it efficient?
1479                 zeroOrManyStepped = false;
1480             }
1481             newLength++;
1482         }
1483         // Строим новую последовательность
1484         zeroOrManyStepped = false;
1485         var newSequence = new ulong[newLength];
1486         long j = 0;
1487         for (var i = 0; i < sequence.Length; i++)
1488         {
1489             //var current = zeroOrManyStepped;
1490             //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1491             //if (current && zeroOrManyStepped)
1492             //    continue;
1493             //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1494             //if (zeroOrManyStepped && newZeroOrManyStepped)
1495             //    continue;
1496             //zeroOrManyStepped = newZeroOrManyStepped;
1497             if (sequence[i] == ZeroOrMany)
1498             {
1499                 if (zeroOrManyStepped)
1500                 {
1501                     continue;
1502                 }
1503                 zeroOrManyStepped = true;
1504             }
1505             else
1506             {
1507                 //if (zeroOrManyStepped) Is it efficient?
1508                 zeroOrManyStepped = false;
1509             }
1510             newSequence[j++] = sequence[i];
1511         }
1512         return newSequence;
1513     }
1514
1515     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1516     public static void TestSimplify()

```

```

1517 {
1518     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
1519         ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1520     var simplifiedSequence = Simplify(sequence);
1521 }
1522 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1523 public List<ulong> GetSimilarSequences() => new List<ulong>();
1524
1525 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1526 public void Prediction()
1527 {
1528     //_links
1529     //_sequences
1530 }
1531
1532 #region From Triplets
1533
1534 //public static void DeleteSequence(Link sequence)
1535 //{
1536 //}
1537
1538 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1539 public List<ulong> CollectMatchingSequences(ulong[] links)
1540 {
1541     if (links.Length == 1)
1542     {
1543         throw new InvalidOperationException("Подпоследовательности с одним элементом не
1544             ↪ поддерживаются.");
1545     }
1546     var leftBound = 0;
1547     var rightBound = links.Length - 1;
1548     var left = links[leftBound++];
1549     var right = links[rightBound--];
1550     var results = new List<ulong>();
1551     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1552     return results;
1553 }
1554
1555 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1556 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1557     ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1558 {
1559     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1560     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1561     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1562     {
1563         var nextLeftLink = middleLinks[leftBound];
1564         var elements = GetRightElements(leftLink, nextLeftLink);
1565         if (leftBound <= rightBound)
1566         {
1567             for (var i = elements.Length - 1; i >= 0; i--)
1568             {
1569                 var element = elements[i];
1570                 if (element != 0)
1571                 {
1572                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
1573                         ↪ rightLink, rightBound, ref results);
1574                 }
1575             }
1576         }
1577         else
1578         {
1579             for (var i = elements.Length - 1; i >= 0; i--)
1580             {
1581                 var element = elements[i];
1582                 if (element != 0)
1583                 {
1584                     results.Add(element);
1585                 }
1586             }
1587         }
1588     }
1589     else
1590     {
1591         var nextRightLink = middleLinks[rightBound];
1592         var elements = GetLeftElements(rightLink, nextRightLink);
1593         if (leftBound <= rightBound)

```

```

1591     {
1592         for (var i = elements.Length - 1; i >= 0; i--)
1593         {
1594             var element = elements[i];
1595             if (element != 0)
1596             {
1597                 CollectMatchingSequences(leftLink, leftBound, middleLinks,
1598                     ↪ elements[i], rightBound - 1, ref results);
1599             }
1600         }
1601     }
1602     else
1603     {
1604         for (var i = elements.Length - 1; i >= 0; i--)
1605         {
1606             var element = elements[i];
1607             if (element != 0)
1608             {
1609                 results.Add(element);
1610             }
1611         }
1612     }
1613 }
1614
1615 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1616 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1617 {
1618     var result = new ulong[5];
1619     TryStepRight(startLink, rightLink, result, 0);
1620     Links.Each(Constants.Any, startLink, couple =>
1621     {
1622         if (couple != startLink)
1623         {
1624             if (TryStepRight(couple, rightLink, result, 2))
1625             {
1626                 return false;
1627             }
1628         }
1629         return true;
1630     });
1631     if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1632     {
1633         result[4] = startLink;
1634     }
1635     return result;
1636 }
1637
1638 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1639 public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1640 {
1641     var added = 0;
1642     Links.Each(startLink, Constants.Any, couple =>
1643     {
1644         if (couple != startLink)
1645         {
1646             var coupleTarget = Links.GetTarget(couple);
1647             if (coupleTarget == rightLink)
1648             {
1649                 result[offset] = couple;
1650                 if (++added == 2)
1651                 {
1652                     return false;
1653                 }
1654             }
1655             else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
1656                 ↪ == Net.And &&
1657             {
1658                 result[offset + 1] = couple;
1659                 if (++added == 2)
1660                 {
1661                     return false;
1662                 }
1663             }
1664         }
1665     });
1666     return added > 0;

```

```

1667     }
1668
1669     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1670     public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1671     {
1672         var result = new ulong[5];
1673         TryStepLeft(startLink, leftLink, result, 0);
1674         Links.Each(startLink, Constants.Any, couple =>
1675         {
1676             if (couple != startLink)
1677             {
1678                 if (TryStepLeft(couple, leftLink, result, 2))
1679                 {
1680                     return false;
1681                 }
1682             }
1683             return true;
1684         });
1685         if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1686         {
1687             result[4] = leftLink;
1688         }
1689         return result;
1690     }
1691
1692     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1693     public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1694     {
1695         var added = 0;
1696         Links.Each(Constants.Any, startLink, couple =>
1697         {
1698             if (couple != startLink)
1699             {
1700                 var coupleSource = Links.GetSource(couple);
1701                 if (coupleSource == leftLink)
1702                 {
1703                     result[offset] = couple;
1704                     if (++added == 2)
1705                     {
1706                         return false;
1707                     }
1708                 }
1709                 else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1710                     ↪ == Net.And &&
1711                 {
1712                     result[offset + 1] = couple;
1713                     if (++added == 2)
1714                     {
1715                         return false;
1716                     }
1717                 }
1718             }
1719             return true;
1720         });
1721         return added > 0;
1722     }
1723
1724     #endregion
1725
1726     #region Walkers
1727
1728     public class PatternMatcher : RightSequenceWalker<ulong>
1729     {
1730         private readonly Sequences _sequences;
1731         private readonly ulong[] _patternSequence;
1732         private readonly HashSet<LinkIndex> _linksInSequence;
1733         private readonly HashSet<LinkIndex> _results;
1734
1735         #region Pattern Match
1736
1737         enum PatternBlockType
1738         {
1739             Undefined,
1740             Gap,
1741             Elements
1742         }
1743
1744         struct PatternBlock
1745         {
1746             public PatternBlockType Type;

```

```

1746         public long Start;
1747         public long Stop;
1748     }
1749
1750     private readonly List<PatternBlock> _pattern;
1751     private int _patternPosition;
1752     private long _sequencePosition;
1753
1754     #endregion
1755
1756     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1757     public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1758         ↳ HashSet<LinkIndex> results)
1759         : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1760     {
1761         _sequences = sequences;
1762         _patternSequence = patternSequence;
1763         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1764             ↳ _sequences.Constants.Any && x != ZeroOrMany));
1765         _results = results;
1766         _pattern = CreateDetailedPattern();
1767     }
1768
1769     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1770     protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
1771         ↳ base.IsElement(link);
1772
1773     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1774     public bool PatternMatch(LinkIndex sequenceToMatch)
1775     {
1776         _patternPosition = 0;
1777         _sequencePosition = 0;
1778         foreach (var part in Walk(sequenceToMatch))
1779         {
1780             if (!PatternMatchCore(part))
1781             {
1782                 break;
1783             }
1784         }
1785         return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1786             ↳ - 1 && _pattern[_patternPosition].Start == 0);
1787     }
1788
1789     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1790     private List<PatternBlock> CreateDetailedPattern()
1791     {
1792         var pattern = new List<PatternBlock>();
1793         var patternBlock = new PatternBlock();
1794         for (var i = 0; i < _patternSequence.Length; i++)
1795         {
1796             if (patternBlock.Type == PatternBlockType.Undefined)
1797             {
1798                 if (_patternSequence[i] == _sequences.Constants.Any)
1799                 {
1800                     patternBlock.Type = PatternBlockType.Gap;
1801                     patternBlock.Start = 1;
1802                     patternBlock.Stop = 1;
1803                 }
1804                 else if (_patternSequence[i] == ZeroOrMany)
1805                 {
1806                     patternBlock.Type = PatternBlockType.Gap;
1807                     patternBlock.Start = 0;
1808                     patternBlock.Stop = long.MaxValue;
1809                 }
1810                 else
1811                 {
1812                     patternBlock.Type = PatternBlockType.Elements;
1813                     patternBlock.Start = i;
1814                     patternBlock.Stop = i;
1815                 }
1816             }
1817             else if (patternBlock.Type == PatternBlockType.Elements)
1818             {
1819                 if (_patternSequence[i] == _sequences.Constants.Any)
1820                 {
1821                     pattern.Add(patternBlock);
1822                     patternBlock = new PatternBlock
1823                     {
1824                         Type = PatternBlockType.Gap,
1825                         Start = 1,

```



```

1822         Stop = 1
1823     };
1824 }
1825 else if (_patternSequence[i] == ZeroOrMany)
1826 {
1827     pattern.Add(patternBlock);
1828     patternBlock = new PatternBlock
1829     {
1830         Type = PatternBlockType.Gap,
1831         Start = 0,
1832         Stop = long.MaxValue
1833     };
1834 }
1835 else
1836 {
1837     patternBlock.Stop = i;
1838 }
1839 }
1840 else // patternBlock.Type == PatternBlockType.Gap
1841 {
1842     if (_patternSequence[i] == _sequences.Constants.Any)
1843     {
1844         patternBlock.Start++;
1845         if (patternBlock.Stop < patternBlock.Start)
1846         {
1847             patternBlock.Stop = patternBlock.Start;
1848         }
1849     }
1850     else if (_patternSequence[i] == ZeroOrMany)
1851     {
1852         patternBlock.Stop = long.MaxValue;
1853     }
1854     else
1855     {
1856         pattern.Add(patternBlock);
1857         patternBlock = new PatternBlock
1858         {
1859             Type = PatternBlockType.Elements,
1860             Start = i,
1861             Stop = i
1862         };
1863     }
1864 }
1865 }
1866 if (patternBlock.Type != PatternBlockType.Undefined)
1867 {
1868     pattern.Add(patternBlock);
1869 }
1870 return pattern;
1871 }
1872
1873 // match: search for regexp anywhere in text
1874 //int match(char* regexp, char* text)
1875 //{
1876 //    do
1877 //    {
1878 //    } while (*text++ != '\0');
1879 //    return 0;
1880 //}
1881
1882 // matchhere: search for regexp at beginning of text
1883 //int matchhere(char* regexp, char* text)
1884 //{
1885 //    if (regexp[0] == '\0')
1886 //        return 1;
1887 //    if (regexp[1] == '*')
1888 //        return matchstar(regexp[0], regexp + 2, text);
1889 //    if (regexp[0] == '$' && regexp[1] == '\0')
1890 //        return *text == '\0';
1891 //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1892 //        return matchhere(regexp + 1, text + 1);
1893 //    return 0;
1894 //}
1895
1896 // matchstar: search for c*regexp at beginning of text
1897 //int matchstar(int c, char* regexp, char* text)
1898 //{
1899 //    do
1900 //    {
1901 //        /* a * matches zero or more instances */

```

```

1901         //         if (matchhere(regexp, text))
1902             //             return 1;
1903         //     } while (*text != '\0' && (*text++ == c || c == '.'));
1904         //     return 0;
1905         //}
1906
1907         //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
1908         //    ↳ long maximumGap)
1909         //{
1910         //     mininumGap = 0;
1911         //     maximumGap = 0;
1912         //     element = 0;
1913         //     for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1914         //     {
1915         //         if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1916         //             mininumGap++;
1917         //         else if (_patternSequence[_patternPosition] == ZeroOrMany)
1918         //             maximumGap = long.MaxValue;
1919         //         else
1920         //             break;
1921         //     }
1922
1923         //     if (maximumGap < mininumGap)
1924         //         maximumGap = mininumGap;
1925         //}
1926
1927         [MethodImpl(MethodImplOptions.AggressiveInlining)]
1928         private bool PatternMatchCore(LinkIndex element)
1929         {
1930             if (_patternPosition >= _pattern.Count)
1931             {
1932                 _patternPosition = -2;
1933                 return false;
1934             }
1935             var currentPatternBlock = _pattern[_patternPosition];
1936             if (currentPatternBlock.Type == PatternBlockType.Gap)
1937             {
1938                 //var currentMatchingBlockLength = (_sequencePosition -
1939                 //    ↳ _lastMatchedBlockPosition);
1940                 if (_sequencePosition < currentPatternBlock.Start)
1941                 {
1942                     _sequencePosition++;
1943                     return true; // Двигаемся дальше
1944                 }
1945                 // Это последний блок
1946                 if (_pattern.Count == _patternPosition + 1)
1947                 {
1948                     _patternPosition++;
1949                     _sequencePosition = 0;
1950                     return false; // Полное соответствие
1951                 }
1952                 else
1953                 {
1954                     if (_sequencePosition > currentPatternBlock.Stop)
1955                     {
1956                         return false; // Соответствие невозможно
1957                     }
1958                     var nextPatternBlock = _pattern[_patternPosition + 1];
1959                     if (_patternSequence[nextPatternBlock.Start] == element)
1960                     {
1961                         if (nextPatternBlock.Start < nextPatternBlock.Stop)
1962                         {
1963                             _patternPosition++;
1964                             _sequencePosition = 1;
1965                         }
1966                         else
1967                         {
1968                             _patternPosition += 2;
1969                             _sequencePosition = 0;
1970                         }
1971                     }
1972                 }
1973             }
1974             else // currentPatternBlock.Type == PatternBlockType.Elements
1975             {
1976                 var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1977                 if (_patternSequence[patternElementPosition] != element)
1978                 {
1979                     return false; // Соответствие невозможно

```

```

1978     }
1979     if (patternElementPosition == currentPatternBlock.Stop)
1980     {
1981         _patternPosition++;
1982         _sequencePosition = 0;
1983     }
1984     else
1985     {
1986         _sequencePosition++;
1987     }
1988 }
1989 return true;
1990 //if (_patternSequence[_patternPosition] != element)
1991 //    return false;
1992 //else
1993 //{
1994 //    _sequencePosition++;
1995 //    _patternPosition++;
1996 //    return true;
1997 //}
1998 ///////////////
1999 //if (_filterPosition == _patternSequence.Length)
2000 //{
2001 //    _filterPosition = -2; // Длиннее чем нужно
2002 //    return false;
2003 //}
2004 //if (element != _patternSequence[_filterPosition])
2005 //{
2006 //    _filterPosition = -1;
2007 //    return false; // Начинается иначе
2008 //}
2009 //filterPosition++;
2010 //if (_filterPosition == (_patternSequence.Length - 1))
2011 //    return false;
2012 //if (_filterPosition >= 0)
2013 //{
2014 //    if (element == _patternSequence[_filterPosition + 1])
2015 //        _filterPosition++;
2016 //    else
2017 //        return false;
2018 //}
2019 //if (_filterPosition < 0)
2020 //{
2021 //    if (element == _patternSequence[0])
2022 //        _filterPosition = 0;
2023 //}
2024 }
2025
2026 [MethodImpl(MethodImplOptions.AggressiveInlining)]
2027 public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
2028 {
2029     foreach (var sequenceToMatch in sequencesToMatch)
2030     {
2031         if (PatternMatch(sequenceToMatch))
2032         {
2033             _results.Add(sequenceToMatch);
2034         }
2035     }
2036 }
2037 }
2038
2039 #endregion
2040 }
2041 }

```

1.93 ./csharp/Platform.Data.Doublets/Sequences/Sequences.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections;
6  using Platform.Collections.Lists;
7  using Platform.Collections.Stacks;
8  using Platform.Threading.Synchronization;
9  using Platform.Data.Doublets.Sequences.Walkers;
10 using LinkIndex = System.UInt64;
11
12 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
13

```

```

14 namespace Platform.Data.Doublets.Sequences
15 {
16     /// <summary>
17     /// Представляет коллекцию последовательностей связей.
18     /// </summary>
19     /// <remarks>
20     /// Обязательно реализовать атомарность каждого публичного метода.
21     ///
22     /// TODO:
23     ///
24     /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
25     /// через естественную группировку по unicode типам, все whitespace вместе, все символы
26     /// ↪ вместе, все числа вместе и т.п.
27     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
28     /// ↪ графа)
29     ///
30     /// х*у - найти все связи между, в последовательностях любой формы, если не стоит
31     /// ↪ ограничитель на то, что является последовательностью, а что нет,
32     /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
33     /// ↪ порядке.
34     ///
35     /// Рост последовательности слева и справа.
36     /// Поиск со звездочкой.
37     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
38     /// так же проблема может быть решена при реализации дистанционных триггеров.
39     /// Нужны ли уникальные указатели вообще?
40     /// Что если обращение к информации будет происходить через содержимое всегда?
41     ///
42     /// Писать тесты.
43     ///
44     ///
45     ///
46     /// Можно убрать зависимость от конкретной реализации Links,
47     /// на зависимость от абстрактного элемента, который может быть представлен несколькими
48     /// ↪ способами.
49     ///
50     /// Можно ли как-то сделать один общий интерфейс
51     ///
52     ///
53     /// Блокчейн и/или гит для распределённой записи транзакций.
54     ///
55     /// </remarks>
56     public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
57     ↪ (после завершения реализации Sequences)
58     {
59         /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
60         /// ↪ связей.</summary>
61         public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
62
63         public SequencesOptions<LinkIndex> Options { get; }
64         public SynchronizedLinks<LinkIndex> Links { get; }
65         private readonly ISynchronization _sync;
66
67         public LinksConstants<LinkIndex> Constants { get; }
68
69         [MethodImpl(MethodImplOptions.AggressiveInlining)]
70         public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
71         {
72             Links = links;
73             _sync = links.SyncRoot;
74             Options = options;
75             Options.ValidateOptions();
76             Options.InitOptions(Links);
77             Constants = links.Constants;
78         }
79
80         [MethodImpl(MethodImplOptions.AggressiveInlining)]
81         public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
82         ↪ SequencesOptions<LinkIndex>()) { }
83
84         [MethodImpl(MethodImplOptions.AggressiveInlining)]
85         public bool IsSequence(LinkIndex sequence)
86         {
87             return _sync.ExecuteReadOperation(() =>
88             {
89                 if (Options.UseSequenceMarker)
90                 {
91                     return Options.MarkedSequenceMatcher.IsMatched(sequence);
92                 }
93             });
94         }
95     }
96 }

```

```

84         return !Links.Unsync.IsPartialPoint(sequence);
85     });
86 }
87
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 private LinkIndex GetSequenceByElements(LinkIndex sequence)
90 {
91     if (Options.UseSequenceMarker)
92     {
93         return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
94     }
95     return sequence;
96 }
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 private LinkIndex GetSequenceElements(LinkIndex sequence)
100 {
101     if (Options.UseSequenceMarker)
102     {
103         var linkContents = new Link<ulong>(Links.GetLink(sequence));
104         if (linkContents.Source == Options.SequenceMarkerLink)
105         {
106             return linkContents.Target;
107         }
108         if (linkContents.Target == Options.SequenceMarkerLink)
109         {
110             return linkContents.Source;
111         }
112     }
113     return sequence;
114 }
115
116 #region Count
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 public LinkIndex Count(ICollection<LinkIndex> restrictions)
120 {
121     if (restrictions.IsNullOrEmpty())
122     {
123         return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
124     }
125     if (restrictions.Count == 1) // Первая связь это адрес
126     {
127         var sequenceIndex = restrictions[0];
128         if (sequenceIndex == Constants.Null)
129         {
130             return 0;
131         }
132         if (sequenceIndex == Constants.Any)
133         {
134             return Count(null);
135         }
136         if (Options.UseSequenceMarker)
137         {
138             return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
139         }
140         return Links.Exists(sequenceIndex) ? 1UL : 0;
141     }
142     throw new NotImplementedException();
143 }
144
145 [MethodImpl(MethodImplOptions.AggressiveInlining)]
146 private LinkIndex CountUsages(params LinkIndex[] restrictions)
147 {
148     if (restrictions.Length == 0)
149     {
150         return 0;
151     }
152     if (restrictions.Length == 1) // Первая связь это адрес
153     {
154         if (restrictions[0] == Constants.Null)
155         {
156             return 0;
157         }
158         var any = Constants.Any;
159         if (Options.UseSequenceMarker)
160         {
161             var elementsLink = GetSequenceElements(restrictions[0]);
162             var sequenceLink = GetSequenceByElements(elementsLink);

```

```

163         if (sequenceLink != Constants.Null)
164         {
165             return Links.Count(any, sequenceLink) + Links.Count(any, elementsLink) -
                ↪ 1;
166         }
167         return Links.Count(any, elementsLink);
168     }
169     return Links.Count(any, restrictions[0]);
170 }
171 throw new NotImplementedException();
172 }
173
174 #endregion
175
176 #region Create
177
178 [MethodImpl(MethodImplOptions.AggressiveInlining)]
179 public LinkIndex Create(ICollection<LinkIndex> restrictions)
180 {
181     return _sync.ExecuteWriteOperation(() =>
182     {
183         if (restrictions.IsNullOrEmpty())
184         {
185             return Constants.Null;
186         }
187         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
188         return CreateCore(restrictions);
189     });
190 }
191
192 [MethodImpl(MethodImplOptions.AggressiveInlining)]
193 private LinkIndex CreateCore(ICollection<LinkIndex> restrictions)
194 {
195     LinkIndex[] sequence = restrictions.SkipFirst();
196     if (Options.UseIndex)
197     {
198         Options.Index.Add(sequence);
199     }
200     var sequenceRoot = default(LinkIndex);
201     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
202     {
203         var matches = Each(restrictions);
204         if (matches.Count > 0)
205         {
206             sequenceRoot = matches[0];
207         }
208     }
209     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
210     {
211         return CompactCore(sequence);
212     }
213     if (sequenceRoot == default)
214     {
215         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
216     }
217     if (Options.UseSequenceMarker)
218     {
219         return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
220     }
221     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
222 }
223
224 #endregion
225
226 #region Each
227
228 [MethodImpl(MethodImplOptions.AggressiveInlining)]
229 public List<LinkIndex> Each(ICollection<LinkIndex> sequence)
230 {
231     var results = new List<LinkIndex>();
232     var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
233     Each(filler.AddFirstAndReturnConstant, sequence);
234     return results;
235 }
236
237 [MethodImpl(MethodImplOptions.AggressiveInlining)]
238 public LinkIndex Each(Func<ICollection<LinkIndex>, LinkIndex> handler, ICollection<LinkIndex>
    ↪ restrictions)
239 {

```

```

240     return _sync.ExecuteReadOperation(() =>
241     {
242         if (restrictions.IsNullOrEmpty())
243         {
244             return Constants.Continue;
245         }
246         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
247         if (restrictions.Count == 1)
248         {
249             var link = restrictions[0];
250             var any = Constants.Any;
251             if (link == any)
252             {
253                 if (Options.UseSequenceMarker)
254                 {
255                     return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
256                                     ↪ Options.SequenceMarkerLink, any));
257                 }
258                 else
259                 {
260                     return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
261                                     ↪ any));
262                 }
263             }
264             if (Options.UseSequenceMarker)
265             {
266                 var sequenceLinkValues = Links.Unsync.GetLink(link);
267                 if (sequenceLinkValues[Constants.SourcePart] ==
268                     ↪ Options.SequenceMarkerLink)
269                 {
270                     link = sequenceLinkValues[Constants.TargetPart];
271                 }
272             }
273             var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
274             sequence[0] = link;
275             return handler(sequence);
276         }
277         else if (restrictions.Count == 2)
278         {
279             throw new NotImplementedException();
280         }
281         else if (restrictions.Count == 3)
282         {
283             return Links.Unsync.Each(handler, restrictions);
284         }
285         else
286         {
287             var sequence = restrictions.SkipFirst();
288             if (Options.UseIndex && !Options.Index.MightContain(sequence))
289             {
290                 return Constants.Break;
291             }
292             return EachCore(handler, sequence);
293         }
294     });
295 }
296
297 [MethodImpl(MethodImplOptions.AggressiveInlining)]
298 private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
299     ↪ values)
300 {
301     var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
302     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
303     ↪ Id.
304     Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
305     ↪ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
306     ↪ matcher.HandleFullMatched;
307     //if (sequence.Length >= 2)
308     if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
309     {
310         return Constants.Break;
311     }
312     var last = values.Count - 2;
313     for (var i = 1; i < last; i++)
314     {
315         if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
316             ↪ Constants.Continue)
317         {
318             return Constants.Break;
319         }
320     }
321     return innerHandler(values);
322 }

```

```

310         return Constants.Break;
311     }
312 }
313 if (values.Count >= 3)
314 {
315     if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
        ↪ != Constants.Continue)
316     {
317         return Constants.Break;
318     }
319 }
320 return Constants.Continue;
321 }
322
323 [MethodImpl(MethodImplOptions.AggressiveInlining)]
324 private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↪ left, LinkIndex right)
325 {
326     return Links.Unsync.Each(doublet =>
327     {
328         var doubletIndex = doublet[Constants.IndexPart];
329         if (StepRight(handler, doubletIndex, right) != Constants.Continue)
330         {
331             return Constants.Break;
332         }
333         if (left != doubletIndex)
334         {
335             return PartialStepRight(handler, doubletIndex, right);
336         }
337         return Constants.Continue;
338     }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
339 }
340
341 [MethodImpl(MethodImplOptions.AggressiveInlining)]
342 private LinkIndex StepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↪ LinkIndex right) => Links.Unsync.Each(rightStep => TryStepRightUp(handler, right,
    ↪ rightStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, left,
    ↪ Constants.Any));
343
344 [MethodImpl(MethodImplOptions.AggressiveInlining)]
345 private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↪ right, LinkIndex stepFrom)
346 {
347     var upStep = stepFrom;
348     var firstSource = Links.Unsync.GetTarget(upStep);
349     while (firstSource != right && firstSource != upStep)
350     {
351         upStep = firstSource;
352         firstSource = Links.Unsync.GetSource(upStep);
353     }
354     if (firstSource == right)
355     {
356         return handler(new LinkAddress<LinkIndex>(stepFrom));
357     }
358     return Constants.Continue;
359 }
360
361 [MethodImpl(MethodImplOptions.AggressiveInlining)]
362 private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↪ LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
    ↪ leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
    ↪ right));
363
364 [MethodImpl(MethodImplOptions.AggressiveInlining)]
365 private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↪ left, LinkIndex stepFrom)
366 {
367     var upStep = stepFrom;
368     var firstTarget = Links.Unsync.GetSource(upStep);
369     while (firstTarget != left && firstTarget != upStep)
370     {
371         upStep = firstTarget;
372         firstTarget = Links.Unsync.GetTarget(upStep);
373     }
374     if (firstTarget == left)
375     {
376         return handler(new LinkAddress<LinkIndex>(stepFrom));
377     }

```



```

378         return Constants.Continue;
379     }
380
381     #endregion
382
383     #region Update
384
385     [MethodImpl(MethodImplOptions.AggressiveInlining)]
386     public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
387     {
388         var sequence = restrictions.SkipFirst();
389         var newSequence = substitution.SkipFirst();
390         if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
391         {
392             return Constants.Null;
393         }
394         if (sequence.IsNullOrEmpty())
395         {
396             return Create(substitution);
397         }
398         if (newSequence.IsNullOrEmpty())
399         {
400             Delete(restrictions);
401             return Constants.Null;
402         }
403         return _sync.ExecuteWriteOperation((Func<ulong>)(() =>
404         {
405             ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
406             Links.EnsureLinkExists(newSequence);
407             return UpdateCore(sequence, newSequence);
408         })));
409     }
410
411     [MethodImpl(MethodImplOptions.AggressiveInlining)]
412     private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence)
413     {
414         LinkIndex bestVariant;
415         if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
416             ↪ !sequence.EqualTo(newSequence))
417         {
418             bestVariant = CompactCore(newSequence);
419         }
420         else
421         {
422             bestVariant = CreateCore(newSequence);
423         }
424         // TODO: Check all options only ones before loop execution
425         // Возможно нужно две версии Each, возвращающий фактические последовательности и с
426         ↪ маркером,
427         // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
428         ↪ можно получить имея только фактические последовательности.
429         foreach (var variant in Each(sequence))
430         {
431             if (variant != bestVariant)
432             {
433                 UpdateOneCore(variant, bestVariant);
434             }
435         }
436         return bestVariant;
437     }
438
439     [MethodImpl(MethodImplOptions.AggressiveInlining)]
440     private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
441     {
442         if (Options.UseGarbageCollection)
443         {
444             var sequenceElements = GetSequenceElements(sequence);
445             var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
446             var sequenceLink = GetSequenceByElements(sequenceElements);
447             var newSequenceElements = GetSequenceElements(newSequence);
448             var newSequenceLink = GetSequenceByElements(newSequenceElements);
449             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
450             {
451                 if (sequenceLink != Constants.Null)
452                 {
453                     Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
454                 }
455                 Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
456             }
457         }
458     }

```

```

        ClearGarbage(sequenceElementsContents.Source);
        ClearGarbage(sequenceElementsContents.Target);
    }
    else
    {
        if (Options.UseSequenceMarker)
        {
            var sequenceElements = GetSequenceElements(sequence);
            var sequenceLink = GetSequenceByElements(sequenceElements);
            var newSequenceElements = GetSequenceElements(newSequence);
            var newSequenceLink = GetSequenceByElements(newSequenceElements);
            if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
            {
                if (sequenceLink != Constants.Null)
                {
                    Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
                }
                Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
            }
        }
        else
        {
            if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
            {
                Links.Unsync.MergeAndDelete(sequence, newSequence);
            }
        }
    }
}

#endregion

#region Delete

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void Delete(IList<LinkIndex> restrictions)
{
    _sync.ExecuteWriteOperation(() =>
    {
        var sequence = restrictions.SkipFirst();
        // TODO: Check all options only ones before loop execution
        foreach (var linkToDelete in Each(sequence))
        {
            DeleteOneCore(linkToDelete);
        }
    });
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
private void DeleteOneCore(LinkIndex link)
{
    if (Options.UseGarbageCollection)
    {
        var sequenceElements = GetSequenceElements(link);
        var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
        var sequenceLink = GetSequenceByElements(sequenceElements);
        if (Options.UseCascadeDelete || CountUsages(link) == 0)
        {
            if (sequenceLink != Constants.Null)
            {
                Links.Unsync.Delete(sequenceLink);
            }
            Links.Unsync.Delete(link);
        }
        ClearGarbage(sequenceElementsContents.Source);
        ClearGarbage(sequenceElementsContents.Target);
    }
    else
    {
        if (Options.UseSequenceMarker)
        {
            var sequenceElements = GetSequenceElements(link);
            var sequenceLink = GetSequenceByElements(sequenceElements);
            if (Options.UseCascadeDelete || CountUsages(link) == 0)
            {
                if (sequenceLink != Constants.Null)
                {
                    Links.Unsync.Delete(sequenceLink);
                }
            }
        }
    }
}

```

```

532     }
533     Links.Unsync.Delete(link);
534 }
535 }
536 else
537 {
538     if (Options.UseCascadeDelete || CountUsages(link) == 0)
539     {
540         Links.Unsync.Delete(link);
541     }
542 }
543 }
544 }
545
546 #endregion
547
548 #region Compactification
549
550 [MethodImpl(MethodImplOptions.AggressiveInlining)]
551 public void CompactAll()
552 {
553     _sync.ExecuteWriteOperation(() =>
554     {
555         var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
556         for (int i = 0; i < sequences.Count; i++)
557         {
558             var sequence = this.ToList(sequences[i]);
559             Compact(sequence.ShiftRight());
560         }
561     });
562 }
563
564 /// <remarks>
565 /// bestVariant можно выбирать по максимальному числу использований,
566 /// но балансированный позволяет гарантировать уникальность (если есть возможность,
567 /// гарантировать его использование в других местах).
568 ///
569 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
570 /// </remarks>
571 [MethodImpl(MethodImplOptions.AggressiveInlining)]
572 public LinkIndex Compact(ICollection<LinkIndex> sequence)
573 {
574     return _sync.ExecuteWriteOperation(() =>
575     {
576         if (sequence.IsNullOrEmpty())
577         {
578             return Constants.Null;
579         }
580         Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
581         return CompactCore(sequence);
582     });
583 }
584
585 [MethodImpl(MethodImplOptions.AggressiveInlining)]
586 private LinkIndex CompactCore(ICollection<LinkIndex> sequence) => UpdateCore(sequence,
587     ↪ sequence);
588
589 #endregion
590
591 #region Garbage Collection
592
593 /// <remarks>
594 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
595 ↪ определить извне или в унаследованном классе
596 /// </remarks>
597 [MethodImpl(MethodImplOptions.AggressiveInlining)]
598 private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
599     ↪ !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
600
601 [MethodImpl(MethodImplOptions.AggressiveInlining)]
602 private void ClearGarbage(LinkIndex link)
603 {
604     if (IsGarbage(link))
605     {
606         var contents = new Link<ulong>(Links.GetLink(link));
607         Links.Unsync.Delete(link);
608         ClearGarbage(contents.Source);
609         ClearGarbage(contents.Target);
610     }
611 }

```

```

}
#endregion

#region Walkers

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
{
    return _sync.ExecuteReadOperation(() =>
    {
        var links = Links.Unsync;
        foreach (var part in Options.Walker.Walk(sequence))
        {
            if (!handler(part))
            {
                return false;
            }
        }
        return true;
    });
}

public class Matcher : RightSequenceWalker<LinkIndex>
{
    private readonly Sequences _sequences;
    private readonly IList<LinkIndex> _patternSequence;
    private readonly HashSet<LinkIndex> _linksInSequence;
    private readonly HashSet<LinkIndex> _results;
    private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
    private readonly HashSet<LinkIndex> _readAsElements;
    private int _filterPosition;

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
        ↳ HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
        ↳ HashSet<LinkIndex> readAsElements = null)
        : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
    {
        _sequences = sequences;
        _patternSequence = patternSequence;
        _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
            ↳ _links.Constants.Any && x != ZeroOrMany));
        _results = results;
        _stopableHandler = stopableHandler;
        _readAsElements = readAsElements;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
        ↳ (_readAsElements != null && _readAsElements.Contains(link)) ||
        ↳ _linksInSequence.Contains(link);

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public bool FullMatch(LinkIndex sequenceToMatch)
    {
        _filterPosition = 0;
        foreach (var part in Walk(sequenceToMatch))
        {
            if (!FullMatchCore(part))
            {
                break;
            }
        }
        return _filterPosition == _patternSequence.Count;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private bool FullMatchCore(LinkIndex element)
    {
        if (_filterPosition == _patternSequence.Count)
        {
            _filterPosition = -2; // Длиннее чем нужно
            return false;
        }
        if (_patternSequence[_filterPosition] != _links.Constants.Any
            && element != _patternSequence[_filterPosition])
        {
            _filterPosition = -1;
            return false; // Начинается/Продолжается иначе
        }
    }
}

```

```

683     }
684     _filterPosition++;
685     return true;
686 }
687
688 [MethodImpl(MethodImplOptions.AggressiveInlining)]
689 public void AddFullMatchedToResults(ICollection<LinkIndex> restrictions)
690 {
691     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
692     if (FullMatch(sequenceToMatch))
693     {
694         _results.Add(sequenceToMatch);
695     }
696 }
697
698 [MethodImpl(MethodImplOptions.AggressiveInlining)]
699 public LinkIndex HandleFullMatched(ICollection<LinkIndex> restrictions)
700 {
701     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
702     if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
703     {
704         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
705     }
706     return _links.Constants.Continue;
707 }
708
709 [MethodImpl(MethodImplOptions.AggressiveInlining)]
710 public LinkIndex HandleFullMatchedSequence(ICollection<LinkIndex> restrictions)
711 {
712     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
713     var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
714     if (sequence != _links.Constants.Null && FullMatch(sequenceToMatch) &&
715         ↪ _results.Add(sequenceToMatch))
716     {
717         return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
718     }
719     return _links.Constants.Continue;
720 }
721
722 /// <remarks>
723 /// TODO: Add support for LinksConstants.Any
724 /// </remarks>
725 [MethodImpl(MethodImplOptions.AggressiveInlining)]
726 public bool PartialMatch(LinkIndex sequenceToMatch)
727 {
728     _filterPosition = -1;
729     foreach (var part in Walk(sequenceToMatch))
730     {
731         if (!PartialMatchCore(part))
732         {
733             break;
734         }
735     }
736     return _filterPosition == _patternSequence.Count - 1;
737 }
738
739 [MethodImpl(MethodImplOptions.AggressiveInlining)]
740 private bool PartialMatchCore(LinkIndex element)
741 {
742     if (_filterPosition == (_patternSequence.Count - 1))
743     {
744         return false; // Нашлось
745     }
746     if (_filterPosition >= 0)
747     {
748         if (element == _patternSequence[_filterPosition + 1])
749         {
750             _filterPosition++;
751         }
752         else
753         {
754             _filterPosition = -1;
755         }
756     }
757     if (_filterPosition < 0)
758     {
759         if (element == _patternSequence[0])
760         {
761             _filterPosition = 0;
762         }
763     }

```

```

761     }
762     }
763     return true; // Ищем дальше
764 }
765
766 [MethodImpl(MethodImplOptions.AggressiveInlining)]
767 public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
768 {
769     if (PartialMatch(sequenceToMatch))
770     {
771         _results.Add(sequenceToMatch);
772     }
773 }
774
775 [MethodImpl(MethodImplOptions.AggressiveInlining)]
776 public LinkIndex HandlePartialMatched(ICollection<LinkIndex> restrictions)
777 {
778     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
779     if (PartialMatch(sequenceToMatch))
780     {
781         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
782     }
783     return _links.Constants.Continue;
784 }
785
786 [MethodImpl(MethodImplOptions.AggressiveInlining)]
787 public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)
788 {
789     foreach (var sequenceToMatch in sequencesToMatch)
790     {
791         if (PartialMatch(sequenceToMatch))
792         {
793             _results.Add(sequenceToMatch);
794         }
795     }
796 }
797
798 [MethodImpl(MethodImplOptions.AggressiveInlining)]
799 public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
800 ↪ sequencesToMatch)
801 {
802     foreach (var sequenceToMatch in sequencesToMatch)
803     {
804         if (PartialMatch(sequenceToMatch))
805         {
806             _readAsElements.Add(sequenceToMatch);
807             _results.Add(sequenceToMatch);
808         }
809     }
810 }
811 #endregion
812 }
813 }
814 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences
8  {
9      public static class SequencesExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static TLink Create<TLink>(this ICollection<TLink> sequences, ICollection<TLink[]>
13         ↪ groupedSequence)
14         {
15             var finalSequence = new TLink[groupedSequence.Count];
16             for (var i = 0; i < finalSequence.Length; i++)
17             {
18                 var part = groupedSequence[i];
19                 finalSequence[i] = part.Length == 1 ? part[0] :
20                 ↪ sequences.Create(part.ShiftRight());
21             }
22             return sequences.Create(finalSequence.ShiftRight());
23         }
24     }
25 }

```

```

21     }
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
25     {
26         var list = new List<TLink>();
27         var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
28         sequences.Each(filler.AddSkipFirstAndReturnConstant, new
29             ↳ LinkAddress<TLink>(sequence));
30         return list;
31     }
32 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Collections.Stacks;
5  using Platform.Converters;
6  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
7  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
8  using Platform.Data.Doublets.Sequences.Converters;
9  using Platform.Data.Doublets.Sequences.Walkers;
10 using Platform.Data.Doublets.Sequences.Indexes;
11 using Platform.Data.Doublets.Sequences.CriterionMatchers;
12 using System.Runtime.CompilerServices;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
19         ↳ ILinks<TLink> must contain GetConstants function.
20     {
21         private static readonly EqualityComparer<TLink> _equalityComparer =
22             ↳ EqualityComparer<TLink>.Default;
23
24         public TLink SequenceMarkerLink
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28             [MethodImpl(MethodImplOptions.AggressiveInlining)]
29             set;
30         }
31
32         public bool UseCascadeUpdate
33         {
34             [MethodImpl(MethodImplOptions.AggressiveInlining)]
35             get;
36             [MethodImpl(MethodImplOptions.AggressiveInlining)]
37             set;
38         }
39
40         public bool UseCascadeDelete
41         {
42             [MethodImpl(MethodImplOptions.AggressiveInlining)]
43             get;
44             [MethodImpl(MethodImplOptions.AggressiveInlining)]
45             set;
46         }
47
48         public bool UseIndex
49         {
50             [MethodImpl(MethodImplOptions.AggressiveInlining)]
51             get;
52             [MethodImpl(MethodImplOptions.AggressiveInlining)]
53             set;
54         } // TODO: Update Index on sequence update/delete.
55
56         public bool UseSequenceMarker
57         {
58             [MethodImpl(MethodImplOptions.AggressiveInlining)]
59             get;
60             [MethodImpl(MethodImplOptions.AggressiveInlining)]
61             set;
62         }
63
64         public bool UseCompression
65         {
66             [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

65         get;
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         set;
68     }
69
70     public bool UseGarbageCollection
71     {
72         [MethodImpl(MethodImplOptions.AggressiveInlining)]
73         get;
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]
75         set;
76     }
77
78     public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting
79     {
80         [MethodImpl(MethodImplOptions.AggressiveInlining)]
81         get;
82         [MethodImpl(MethodImplOptions.AggressiveInlining)]
83         set;
84     }
85
86     public bool EnforceSingleSequenceVersionOnWriteBasedOnNew
87     {
88         [MethodImpl(MethodImplOptions.AggressiveInlining)]
89         get;
90         [MethodImpl(MethodImplOptions.AggressiveInlining)]
91         set;
92     }
93
94     public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher
95     {
96         [MethodImpl(MethodImplOptions.AggressiveInlining)]
97         get;
98         [MethodImpl(MethodImplOptions.AggressiveInlining)]
99         set;
100     }
101
102     public IConverter<IList<TLink>, TLink> LinksToSequenceConverter
103     {
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         get;
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         set;
108     }
109
110     public ISequenceIndex<TLink> Index
111     {
112         [MethodImpl(MethodImplOptions.AggressiveInlining)]
113         get;
114         [MethodImpl(MethodImplOptions.AggressiveInlining)]
115         set;
116     }
117
118     public ISequenceWalker<TLink> Walker
119     {
120         [MethodImpl(MethodImplOptions.AggressiveInlining)]
121         get;
122         [MethodImpl(MethodImplOptions.AggressiveInlining)]
123         set;
124     }
125
126     public bool ReadFullSequence
127     {
128         [MethodImpl(MethodImplOptions.AggressiveInlining)]
129         get;
130         [MethodImpl(MethodImplOptions.AggressiveInlining)]
131         set;
132     }
133
134     // TODO: Реализовать компактификацию при чтении
135     //public bool EnforceSingleSequenceVersionOnRead { get; set; }
136     //public bool UseRequestMarker { get; set; }
137     //public bool StoreRequestResults { get; set; }
138
139     [MethodImpl(MethodImplOptions.AggressiveInlining)]
140     public void InitOptions(ISynchronizedLinks<TLink> links)
141     {
142         if (UseSequenceMarker)
143         {
144             if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
145             {

```



```

146         SequenceMarkerLink = links.CreatePoint();
147     }
148     else
149     {
150         if (!links.Exists(SequenceMarkerLink))
151         {
152             var link = links.CreatePoint();
153             if (!_equalityComparer.Equals(link, SequenceMarkerLink))
154             {
155                 throw new InvalidOperationException("Cannot recreate sequence marker
156                     ↪ link.");
157             }
158         }
159         if (MarkedSequenceMatcher == null)
160         {
161             MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
162                 ↪ SequenceMarkerLink);
163         }
164         var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
165         if (UseCompression)
166         {
167             if (LinksToSequenceConverter == null)
168             {
169                 ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
170                 if (UseSequenceMarker)
171                 {
172                     totalSequenceSymbolFrequencyCounter = new
173                         ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
174                             ↪ MarkedSequenceMatcher);
175                 }
176                 else
177                 {
178                     totalSequenceSymbolFrequencyCounter = new
179                         ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
180                 }
181                 var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
182                     ↪ totalSequenceSymbolFrequencyCounter);
183                 var compressingConverter = new CompressingConverter<TLink>(links,
184                     ↪ balancedVariantConverter, doubletFrequenciesCache);
185                 LinksToSequenceConverter = compressingConverter;
186             }
187         }
188         else
189         {
190             if (LinksToSequenceConverter == null)
191             {
192                 LinksToSequenceConverter = balancedVariantConverter;
193             }
194         }
195         if (UseIndex && Index == null)
196         {
197             Index = new SequenceIndex<TLink>(links);
198         }
199         if (Walker == null)
200         {
201             Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
202         }
203     }
204 }
205
206 [MethodImpl(MethodImplOptions.AggressiveInlining)]
207 public void ValidateOptions()
208 {
209     if (UseGarbageCollection && !UseSequenceMarker)
210     {
211         throw new NotSupportedException("To use garbage collection UseSequenceMarker
212             ↪ option must be on.");
213     }
214 }

```

1.96 ./csharp/Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs

```

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

```

```

5
6 namespace Platform.Data.Doublets.Sequences.Walkers
7 {
8     public interface ISequenceWalker<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         IEnumerable<TLink> Walk(TLink sequence);
12     }
13 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Collections.Stacks;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Sequences.Walkers
9 {
10     public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
14             ⇒ isElement) : base(links, stack, isElement) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
18             ⇒ links.IsPartialPoint) { }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override TLink GetNextElementAfterPop(TLink element) =>
22             ⇒ _links.GetSource(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override TLink GetNextElementAfterPush(TLink element) =>
26             ⇒ _links.GetTarget(element);
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override IEnumerable<TLink> WalkContents(TLink element)
30         {
31             var links = _links;
32             var parts = links.GetLink(element);
33             var start = links.Constants.SourcePart;
34             for (var i = parts.Count - 1; i >= start; i--)
35             {
36                 var part = parts[i];
37                 if (IsElement(part))
38                 {
39                     yield return part;
40                 }
41             }
42         }
43     }
44 }

```

1.98 ./csharp/Platform.Data.Doublets.Sequences.Walkers/LeveledSequenceWalker.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 // #define USEARRAYPOOL
8 #if USEARRAYPOOL
9 using Platform.Collections;
10 #endif
11
12 namespace Platform.Data.Doublets.Sequences.Walkers
13 {
14     public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
15     {
16         private static readonly EqualityComparer<TLink> _equalityComparer =
17             ⇒ EqualityComparer<TLink>.Default;
18
19         private readonly Func<TLink, bool> _isElement;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
23             ⇒ base(links) => _isElement = isElement;
24     }
25 }

```

```

22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
24         ↪ _links.IsPartialPoint;
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public TLink[] ToArray(TLink sequence)
31     {
32         var length = 1;
33         var array = new TLink[length];
34         array[0] = sequence;
35         if (_isElement(sequence))
36         {
37             return array;
38         }
39         bool hasElements;
40         do
41         {
42             length *= 2;
43             #if USEARRAYPOOL
44                 var nextArray = ArrayPool.Allocate<ulong>(length);
45             #else
46                 var nextArray = new TLink[length];
47             #endif
48             hasElements = false;
49             for (var i = 0; i < array.Length; i++)
50             {
51                 var candidate = array[i];
52                 if (_equalityComparer.Equals(array[i], default))
53                 {
54                     continue;
55                 }
56                 var doubletOffset = i * 2;
57                 if (_isElement(candidate))
58                 {
59                     nextArray[doubletOffset] = candidate;
60                 }
61                 else
62                 {
63                     var links = _links;
64                     var link = links.GetLink(candidate);
65                     var linkSource = links.GetSource(link);
66                     var linkTarget = links.GetTarget(link);
67                     nextArray[doubletOffset] = linkSource;
68                     nextArray[doubletOffset + 1] = linkTarget;
69                     if (!hasElements)
70                     {
71                         hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
72                     }
73                 }
74             }
75             #if USEARRAYPOOL
76                 if (array.Length > 1)
77                 {
78                     ArrayPool.Free(array);
79                 }
80             #endif
81             array = nextArray;
82         }
83         while (hasElements);
84         var filledElementsCount = CountFilledElements(array);
85         if (filledElementsCount == array.Length)
86         {
87             return array;
88         }
89         else
90         {
91             return CopyFilledElements(array, filledElementsCount);
92         }
93     }
94
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
97     {
98         var finalArray = new TLink[filledElementsCount];
99         for (int i = 0, j = 0; i < array.Length; i++)

```

```

100     {
101         if (!_equalityComparer.Equals(array[i], default))
102         {
103             finalArray[j] = array[i];
104             j++;
105         }
106     }
107     #if USEARRAYPOOL
108         ArrayPool.Free(array);
109     #endif
110     return finalArray;
111 }
112
113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
114 private static int CountFilledElements(TLink[] array)
115 {
116     var count = 0;
117     for (var i = 0; i < array.Length; i++)
118     {
119         if (!_equalityComparer.Equals(array[i], default))
120         {
121             count++;
122         }
123     }
124     return count;
125 }
126 }
127 }

```

1.99 ./csharp/Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Stacks;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.Walkers
9  {
10     public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
14             ↪ isElement) : base(links, stack, isElement) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
18             ↪ stack, links.IsPartialPoint) { }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override TLink GetNextElementAfterPop(TLink element) =>
22             ↪ _links.GetTarget(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override TLink GetNextElementAfterPush(TLink element) =>
26             ↪ _links.GetSource(element);
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override IEnumerable<TLink> WalkContents(TLink element)
30         {
31             var parts = _links.GetLink(element);
32             for (var i = _links.Constants.SourcePart; i < parts.Count; i++)
33             {
34                 var part = parts[i];
35                 if (IsElement(part))
36                 {
37                     yield return part;
38                 }
39             }
40         }
41     }
42 }

```

1.100 ./csharp/Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Stacks;
5

```

```

6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.Walkers
9  {
10     public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
11         ↳ ISequenceWalker<TLink>
12     {
13         private readonly IStack<TLink> _stack;
14         private readonly Func<TLink, bool> _isElement;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
18             ↳ isElement) : base(links)
19         {
20             _stack = stack;
21             _isElement = isElement;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
26             ↳ stack, links.IsPartialPoint) { }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public IEnumerable<TLink> Walk(TLink sequence)
30         {
31             _stack.Clear();
32             var element = sequence;
33             if (IsElement(element))
34             {
35                 yield return element;
36             }
37             else
38             {
39                 while (true)
40                 {
41                     if (IsElement(element))
42                     {
43                         if (_stack.IsEmpty)
44                         {
45                             break;
46                         }
47                         element = _stack.Pop();
48                         foreach (var output in WalkContents(element))
49                         {
50                             yield return output;
51                         }
52                         element = GetNextElementAfterPop(element);
53                     }
54                     else
55                     {
56                         _stack.Push(element);
57                         element = GetNextElementAfterPush(element);
58                     }
59                 }
60             }
61         }
62
63         [MethodImpl(MethodImplOptions.AggressiveInlining)]
64         protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         protected abstract TLink GetNextElementAfterPop(TLink element);
68
69         [MethodImpl(MethodImplOptions.AggressiveInlining)]
70         protected abstract TLink GetNextElementAfterPush(TLink element);
71
72         [MethodImpl(MethodImplOptions.AggressiveInlining)]
73         protected abstract IEnumerable<TLink> WalkContents(TLink element);
74     }
75 }

```

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

```

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

```

```

9     public class Stack<TLink> : LinksOperatorBase<TLink>, IStack<TLink>
10    {
11        private static readonly EqualityComparer<TLink> _equalityComparer =
12            ↪ EqualityComparer<TLink>.Default;
13
14        private readonly TLink _stack;
15
16        public bool IsEmpty
17        {
18            [MethodImpl(MethodImplOptions.AggressiveInlining)]
19            get => _equalityComparer.Equals(Peek(), _stack);
20        }
21
22        [MethodImpl(MethodImplOptions.AggressiveInlining)]
23        public Stack(ILinks<TLink> links, TLink stack) : base(links) => _stack = stack;
24
25        [MethodImpl(MethodImplOptions.AggressiveInlining)]
26        private TLink GetStackMarker() => _links.GetSource(_stack);
27
28        [MethodImpl(MethodImplOptions.AggressiveInlining)]
29        private TLink GetTop() => _links.GetTarget(_stack);
30
31        [MethodImpl(MethodImplOptions.AggressiveInlining)]
32        public TLink Peek() => _links.GetTarget(GetTop());
33
34        [MethodImpl(MethodImplOptions.AggressiveInlining)]
35        public TLink Pop()
36        {
37            var element = Peek();
38            if (!_equalityComparer.Equals(element, _stack))
39            {
40                var top = GetTop();
41                var previousTop = _links.GetSource(top);
42                _links.Update(_stack, GetStackMarker(), previousTop);
43                _links.Delete(top);
44            }
45            return element;
46        }
47
48        [MethodImpl(MethodImplOptions.AggressiveInlining)]
49        public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
50            ↪ _links.GetOrCreate(GetTop(), element));
51    }

```

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

```

1     using System.Runtime.CompilerServices;
2
3     #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5     namespace Platform.Data.Doublets.Stacks
6     {
7         public static class StackExtensions
8         {
9             [MethodImpl(MethodImplOptions.AggressiveInlining)]
10            public static TLink CreateStack<TLink>(this ILinks<TLink> links, TLink stackMarker)
11            {
12                var stackPoint = links.CreatePoint();
13                var stack = links.Update(stackPoint, stackMarker, stackPoint);
14                return stack;
15            }
16        }
17    }

```

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

```

1     using System;
2     using System.Collections.Generic;
3     using System.Runtime.CompilerServices;
4     using Platform.Data.Doublets;
5     using Platform.Threading.Synchronization;
6
7     #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9     namespace Platform.Data.Doublets
10    {
11        /// <remarks>
12        /// TODO: Autogeneration of synchronized wrapper (decorator).
13        /// TODO: Try to unfold code of each method using IL generation for performance improvements.
14        /// TODO: Or even to unfold multiple layers of implementations.

```

```

15  /// </remarks>
16  public class SynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress>
17  {
18      public LinksConstants<TLinkAddress> Constants
19      {
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          get;
22      }
23
24      public ISynchronization SyncRoot
25      {
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          get;
28      }
29
30      public ILinks<TLinkAddress> Sync
31      {
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          get;
34      }
35
36      public ILinks<TLinkAddress> Unsync
37      {
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          get;
40      }
41
42      [MethodImpl(MethodImplOptions.AggressiveInlining)]
43      public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
44          ↳ ReaderWriterLockSynchronization(), links) { }
45
46      [MethodImpl(MethodImplOptions.AggressiveInlining)]
47      public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
48      {
49          SyncRoot = synchronization;
50          Sync = this;
51          Unsync = links;
52          Constants = links.Constants;
53      }
54
55      [MethodImpl(MethodImplOptions.AggressiveInlining)]
56      public TLinkAddress Count(IList<TLinkAddress> restriction) =>
57          ↳ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
58
59      [MethodImpl(MethodImplOptions.AggressiveInlining)]
60      public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
61          ↳ IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
62          ↳ restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
63
64      [MethodImpl(MethodImplOptions.AggressiveInlining)]
65      public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
66          ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
67
68      [MethodImpl(MethodImplOptions.AggressiveInlining)]
69      public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
70          ↳ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
71          ↳ Unsync.Update);
72
73      [MethodImpl(MethodImplOptions.AggressiveInlining)]
74      public void Delete(IList<TLinkAddress> restrictions) =>
75          ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
76
77      //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
78          ↳ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
79      //{
80      //    if (restriction != null && substitution != null &&
81          ↳ !substitution.EqualTo(restriction))
82          //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
83          ↳ substitution, substitutedHandler, Unsync.Trigger);
84      //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
85          ↳ substitutedHandler, Unsync.Trigger);
86      //}
87  }

```

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

```

1  using System;
2  using System.Text;

```

```

3 using System.Collections.Generic;
4 using System.Runtime.CompilerServices;
5 using Platform.Singletons;
6 using Platform.Data.Doublets.Unicode;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets
11 {
12     public static class UInt64LinksExtensions
13     {
14         public static readonly LinksConstants<ulong> Constants =
15             ↳ Default<LinksConstants<ulong>>.Instance;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
22         {
23             if (sequence == null)
24             {
25                 return false;
26             }
27             var constants = links.Constants;
28             for (var i = 0; i < sequence.Length; i++)
29             {
30                 if (sequence[i] == constants.Any)
31                 {
32                     return true;
33                 }
34             }
35             return false;
36         }
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
40             ↳ Func<Link<ulong>, bool> isElement, bool renderIndex = false, bool renderDebug =
41             ↳ false)
42         {
43             var sb = new StringBuilder();
44             var visited = new HashSet<ulong>();
45             links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
46                 ↳ innerSb.Append(link.Index), renderIndex, renderDebug);
47             return sb.ToString();
48         }
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
52             ↳ Func<Link<ulong>, bool> isElement, Action<StringBuilder, Link<ulong>> appendElement,
53             ↳ bool renderIndex = false, bool renderDebug = false)
54         {
55             var sb = new StringBuilder();
56             var visited = new HashSet<ulong>();
57             links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
58                 ↳ renderDebug);
59             return sb.ToString();
60         }
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
64             ↳ HashSet<ulong> visited, ulong linkIndex, Func<Link<ulong>, bool> isElement,
65             ↳ Action<StringBuilder, Link<ulong>> appendElement, bool renderIndex = false, bool
66             ↳ renderDebug = false)
67         {
68             if (sb == null)
69             {
70                 throw new ArgumentNullException(nameof(sb));
71             }
72             if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
73                 ↳ Constants.Itself)
74             {
75                 return;
76             }
77             if (links.Exists(linkIndex))
78             {
79                 if (visited.Add(linkIndex))
80                 {
81                     sb.Append('(');
82                 }
83             }
84         }
85     }
86 }

```



```

71     var link = new Link<ulong>(links.GetLink(linkIndex));
72     if (renderIndex)
73     {
74         sb.Append(link.Index);
75         sb.Append(':');
76     }
77     if (link.Source == link.Index)
78     {
79         sb.Append(link.Index);
80     }
81     else
82     {
83         var source = new Link<ulong>(links.GetLink(link.Source));
84         if (isElement(source))
85         {
86             appendElement(sb, source);
87         }
88         else
89         {
90             links.AppendStructure(sb, visited, source.Index, isElement,
91                 ↪ appendElement, renderIndex);
92         }
93     }
94     sb.Append(' ');
95     if (link.Target == link.Index)
96     {
97         sb.Append(link.Index);
98     }
99     else
100    {
101        var target = new Link<ulong>(links.GetLink(link.Target));
102        if (isElement(target))
103        {
104            appendElement(sb, target);
105        }
106        else
107        {
108            links.AppendStructure(sb, visited, target.Index, isElement,
109                ↪ appendElement, renderIndex);
110        }
111    }
112    sb.Append(')');
113    }
114    else
115    {
116        if (renderDebug)
117        {
118            sb.Append('*');
119        }
120        sb.Append(linkIndex);
121    }
122    }
123    }
124    }
125    }
126    }
127    }
128    }
129    }
130    }
131    }

```

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

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using System.IO;
5  using System.Runtime.CompilerServices;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Platform.Disposables;
9  using Platform.Timestamps;
10 using Platform.Unsafe;
11 using Platform.IO;
12 using Platform.Data.Doublets.Decorators;
13 using Platform.Exceptions;
14

```

```

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

```

```

93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
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     [MethodImpl(MethodImplOptions.AggressiveInlining)]
104     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
105         ↳ transactionId, Link<ulong> before) : this(uniqueTimestampFactory, transactionId,
106         ↳ before, default) { }
107
108     [MethodImpl(MethodImplOptions.AggressiveInlining)]
109     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
110         ↳ transactionId) : this(uniqueTimestampFactory, transactionId, default, default) {
111         ↳ }
112
113     [MethodImpl(MethodImplOptions.AggressiveInlining)]
114     public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
115         ↳ {After}";
116
117     [MethodImpl(MethodImplOptions.AggressiveInlining)]
118     public override bool Equals(object obj) => obj is Transition transition ?
119         ↳ Equals(transition) : false;
120
121     [MethodImpl(MethodImplOptions.AggressiveInlining)]
122     public override int GetHashCode() => (TransactionId, Before, After,
123         ↳ Timestamp).GetHashCode();
124
125     [MethodImpl(MethodImplOptions.AggressiveInlining)]
126     public bool Equals(Transition other) => TransactionId == other.TransactionId &&
127         ↳ Before == other.Before && After == other.After && Timestamp == other.Timestamp;
128
129     [MethodImpl(MethodImplOptions.AggressiveInlining)]
130     public static bool operator ==(Transition left, Transition right) =>
131         ↳ left.Equals(right);
132
133     [MethodImpl(MethodImplOptions.AggressiveInlining)]
134     public static bool operator !=(Transition left, Transition right) => !(left ==
135         ↳ right);
136 }
137
138 /// <remarks>
139 /// Другие варианты реализации транзакций (атомарности):
140 /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
141   ↳ Target)) и индексов.
142 /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
143   ↳ потребуется решить вопрос
144 /// со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
145   ↳ пересечениями идентификаторов.
146 ///
147 /// Где хранить промежуточный список транзакций?
148 ///
149 /// В оперативной памяти:
150 /// Минусы:
151 /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
152   ↳ так как нужно отдельно выделять память под список трансформаций.
153 /// 2. Выделенной оперативной памяти может не хватить, в том случае,
154   ↳ если транзакция использует слишком много трансформаций.
155 /// -> Можно использовать жёсткий диск для слишком длинных транзакций.
156 /// -> Максимальный размер списка трансформаций можно ограничить / задать
157   ↳ константой.
158 /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
159   ↳ создавая задержку.
160 ///
161 /// На жёстком диске:
162 /// Минусы:
163 /// 1. Длительный отклик, на запись каждой трансформации.
164 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
165   ↳ -> Это может решаться упаковкой/исключением дублирующих операций.
166   ↳ -> Также это может решаться тем, что короткие транзакции вообще
167       ↳ не будут записываться в случае отката.
168 /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
169   ↳ операции (трансформации)

```

```

154     ///      будут записаны в лог.
155     ///
156     /// </remarks>
157     public class Transaction : DisposableBase
158     {
159         private readonly Queue<Transition> _transitions;
160         private readonly UInt64LinksTransactionsLayer _layer;
161         public bool IsCommitted { get; private set; }
162         public bool IsReverted { get; private set; }
163
164         [MethodImpl(MethodImplOptions.AggressiveInlining)]
165         public Transaction(UInt64LinksTransactionsLayer layer)
166         {
167             _layer = layer;
168             if (_layer._currentTransactionId != 0)
169             {
170                 throw new NotSupportedException("Nested transactions not supported.");
171             }
172             IsCommitted = false;
173             IsReverted = false;
174             _transitions = new Queue<Transition>();
175             SetCurrentTransaction(layer, this);
176         }
177
178         [MethodImpl(MethodImplOptions.AggressiveInlining)]
179         public void Commit()
180         {
181             EnsureTransactionAllowsWriteOperations(this);
182             while (_transitions.Count > 0)
183             {
184                 var transition = _transitions.Dequeue();
185                 _layer._transitions.Enqueue(transition);
186             }
187             _layer._lastCommittedTransactionId = _layer._currentTransactionId;
188             IsCommitted = true;
189         }
190
191         [MethodImpl(MethodImplOptions.AggressiveInlining)]
192         private void Revert()
193         {
194             EnsureTransactionAllowsWriteOperations(this);
195             var transitionsToRevert = new Transition[_transitions.Count];
196             _transitions.CopyTo(transitionsToRevert, 0);
197             for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
198             {
199                 _layer.RevertTransition(transitionsToRevert[i]);
200             }
201             IsReverted = true;
202         }
203
204         [MethodImpl(MethodImplOptions.AggressiveInlining)]
205         public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
206             → Transaction transaction)
207         {
208             layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
209             layer._currentTransactionTransitions = transaction._transitions;
210             layer._currentTransaction = transaction;
211         }
212
213         [MethodImpl(MethodImplOptions.AggressiveInlining)]
214         public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
215         {
216             if (transaction.IsReverted)
217             {
218                 throw new InvalidOperationException("Transation is reverted.");
219             }
220             if (transaction.IsCommitted)
221             {
222                 throw new InvalidOperationException("Transation is committed.");
223             }
224         }
225
226         [MethodImpl(MethodImplOptions.AggressiveInlining)]
227         protected override void Dispose(bool manual, bool wasDisposed)
228         {
229             if (!wasDisposed && _layer != null && !_layer.Disposable.IsDisposed)
230             {
231                 if (!IsCommitted && !IsReverted)
232                 {

```

```

232         Revert();
233     }
234     _layer.ResetCurrentTransation();
235 }
236 }
237 }
238
239 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
240
241 private readonly string _logAddress;
242 private readonly FileStream _log;
243 private readonly Queue<Transition> _transitions;
244 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
245 private Task _transitionsPusher;
246 private Transition _lastCommittedTransition;
247 private ulong _currentTransactionId;
248 private Queue<Transition> _currentTransactionTransitions;
249 private Transaction _currentTransaction;
250 private ulong _lastCommittedTransactionId;
251
252 [MethodImpl(MethodImplOptions.AggressiveInlining)]
253 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
254     : base(links)
255 {
256     if (string.IsNullOrEmpty(logAddress))
257     {
258         throw new ArgumentNullException(nameof(logAddress));
259     }
260     // В первой строке файла хранится последняя закоммиченную транзакцию.
261     // При запуске это используется для проверки удачного закрытия файла лога.
262     // In the first line of the file the last committed transaction is stored.
263     // On startup, this is used to check that the log file is successfully closed.
264     var lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
265     var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
266     if (!lastCommittedTransition.Equals(lastWrittenTransition))
267     {
268         Dispose();
269         throw new NotSupportedException("Database is damaged, autorecovery is not
270             ↳ supported yet.");
271     }
272     if (lastCommittedTransition == default)
273     {
274         FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
275     }
276     _lastCommittedTransition = lastCommittedTransition;
277     // TODO: Think about a better way to calculate or store this value
278     var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
279     _lastCommittedTransactionId = allTransitions.Length > 0 ? allTransitions.Max(x =>
280         ↳ x.TransactionId) : 0;
281     _uniqueTimestampFactory = new UniqueTimestampFactory();
282     _logAddress = logAddress;
283     _log = FileHelpers.Append(logAddress);
284     _transitions = new Queue<Transition>();
285     _transitionsPusher = new Task(TransitionsPusher);
286     _transitionsPusher.Start();
287 }
288
289 [MethodImpl(MethodImplOptions.AggressiveInlining)]
290 public IList<ulong> GetLinkValue(ulong link) => _links.GetLink(link);
291
292 [MethodImpl(MethodImplOptions.AggressiveInlining)]
293 public override ulong Create(IList<ulong> restrictions)
294 {
295     var createdLinkIndex = _links.Create();
296     var createdLink = new Link<ulong>(_links.GetLink(createdLinkIndex));
297     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
298         ↳ default, createdLink));
299     return createdLinkIndex;
300 }
301
302 [MethodImpl(MethodImplOptions.AggressiveInlining)]
303 public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
304 {
305     var linkIndex = restrictions[_constants.IndexPart];
306     var beforeLink = new Link<ulong>(_links.GetLink(linkIndex));
307     linkIndex = _links.Update(restrictions, substitution);
308     var afterLink = new Link<ulong>(_links.GetLink(linkIndex));
309     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
310         ↳ beforeLink, afterLink));

```

```

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

```

```

382     }
383
384     [MethodImpl(MethodImplOptions.AggressiveInlining)]
385     public Transaction BeginTransaction() => new Transaction(this);
386
387     [MethodImpl(MethodImplOptions.AggressiveInlining)]
388     private void DisposeTransitions()
389     {
390         try
391         {
392             var pusher = _transitionsPusher;
393             if (pusher != null)
394             {
395                 _transitionsPusher = null;
396                 pusher.Wait();
397             }
398             if (_transitions != null)
399             {
400                 PushTransitions();
401             }
402             _log.DisposeIfPossible();
403             FileHelpers.WriteFirst(_logAddress, _lastCommittedTransition);
404         }
405         catch (Exception ex)
406         {
407             ex.Ignore();
408         }
409     }
410
411     #region DisposalBase
412
413     [MethodImpl(MethodImplOptions.AggressiveInlining)]
414     protected override void Dispose(bool manual, bool wasDisposed)
415     {
416         if (!wasDisposed)
417         {
418             DisposeTransitions();
419         }
420         base.Dispose(manual, wasDisposed);
421     }
422
423     #endregion
424 }
425 }

```

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

```

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

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Converters;
4  using Platform.Data.Doublets.Sequences.Indexes;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Unicode
9  {
10     public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
        ↳ 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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
        ↳ charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
        ↳ TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
19         {
20             _charToUnicodeSymbolConverter = charToUnicodeSymbolConverter;
21             _index = index;
22             _listToSequenceLinkConverter = listToSequenceLinkConverter;
23             _unicodeSequenceMarker = unicodeSequenceMarker;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Convert(string source)
28         {
29             var elements = new TLink[source.Length];
30             for (int i = 0; i < elements.Length; i++)
31             {
32                 elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
33             }
34             _index.Add(elements);
35             var sequence = _listToSequenceLinkConverter.Convert(elements);
36             return _links.GetOrCreate(sequence, _unicodeSequenceMarker);
37         }
38     }
39 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Globalization;
4  using System.Runtime.CompilerServices;
5  using System.Text;
6  using Platform.Data.Sequences;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Unicode
11 {
12     public class UnicodeMap
13     {
14         public static readonly ulong FirstCharLink = 1;
15         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
16         public static readonly ulong MapSize = 1 + char.MaxValue;
17
18         private readonly ILinks<ulong> _links;
19         private bool _initialized;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public UnicodeMap(ILinks<ulong> links) => _links = links;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public static UnicodeMap InitNew(ILinks<ulong> links)
26         {
27             var map = new UnicodeMap(links);
28             map.Init();
29             return map;
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public void Init()
34         {
35             if (!_initialized)
36             {

```



```

37         return;
38     }
39     _initialized = true;
40     var firstLink = _links.CreatePoint();
41     if (firstLink != FirstCharLink)
42     {
43         _links.Delete(firstLink);
44     }
45     else
46     {
47         for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
48         {
49             // From NIL to It (NIL -> Character) transformation meaning, (or infinite
50             ↪ amount of NIL characters before actual Character)
51             var createdLink = _links.CreatePoint();
52             _links.Update(createdLink, firstLink, createdLink);
53             if (createdLink != i)
54             {
55                 throw new InvalidOperationException("Unable to initialize UTF 16
56                 ↪ table.");
57             }
58         }
59     }
60     // 0 - null link
61     // 1 - nil character (0 character)
62     // ...
63     // 65536 (0(1) + 65535 = 65536 possible values)
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     public static ulong FromCharToLink(char character) => (ulong)character + 1;
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     public static char FromLinkToChar(ulong link) => (char)(link - 1);
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     public static bool IsCharLink(ulong link) => link <= MapSize;
73
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     public static string FromLinksToString(IList<ulong> linksList)
76     {
77         var sb = new StringBuilder();
78         for (int i = 0; i < linksList.Count; i++)
79         {
80             sb.Append(FromLinkToChar(linksList[i]));
81         }
82         return sb.ToString();
83     }
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
87     {
88         var sb = new StringBuilder();
89         if (links.Exists(link))
90         {
91             StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
92             ↪ x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
93             ↪ element =>
94             {
95                 sb.Append(FromLinkToChar(element));
96                 return true;
97             }
98         }
99         return sb.ToString();
100     }
101
102     [MethodImpl(MethodImplOptions.AggressiveInlining)]
103     public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
104     ↪ chars.Length);
105
106     [MethodImpl(MethodImplOptions.AggressiveInlining)]
107     public static ulong[] FromCharsToLinkArray(char[] chars, int count)
108     {
109         // char array to ulong array
110         var linksSequence = new ulong[count];
111         for (var i = 0; i < count; i++)
112         {

```

```

111         linksSequence[i] = FromCharToLink(chars[i]);
112     }
113     return linksSequence;
114 }
115
116 [MethodImpl(MethodImplOptions.AggressiveInlining)]
117 public static ulong[] FromStringToLinkArray(string sequence)
118 {
119     // char array to ulong array
120     var linksSequence = new ulong[sequence.Length];
121     for (var i = 0; i < sequence.Length; i++)
122     {
123         linksSequence[i] = FromCharToLink(sequence[i]);
124     }
125     return linksSequence;
126 }
127
128 [MethodImpl(MethodImplOptions.AggressiveInlining)]
129 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
130 {
131     var result = new List<ulong[]>();
132     var offset = 0;
133     while (offset < sequence.Length)
134     {
135         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
136         var relativeLength = 1;
137         var absoluteLength = offset + relativeLength;
138         while (absoluteLength < sequence.Length &&
139             currentCategory ==
140                 CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
141         {
142             relativeLength++;
143             absoluteLength++;
144         }
145         // char array to ulong array
146         var innerSequence = new ulong[relativeLength];
147         var maxLength = offset + relativeLength;
148         for (var i = offset; i < maxLength; i++)
149         {
150             innerSequence[i - offset] = FromCharToLink(sequence[i]);
151         }
152         result.Add(innerSequence);
153         offset += relativeLength;
154     }
155     return result;
156 }
157
158 [MethodImpl(MethodImplOptions.AggressiveInlining)]
159 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
160 {
161     var result = new List<ulong[]>();
162     var offset = 0;
163     while (offset < array.Length)
164     {
165         var relativeLength = 1;
166         if (array[offset] <= LastCharLink)
167         {
168             var currentCategory =
169                 CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
170             var absoluteLength = offset + relativeLength;
171             while (absoluteLength < array.Length &&
172                 array[absoluteLength] <= LastCharLink &&
173                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
174                     array[absoluteLength])))
175             {
176                 relativeLength++;
177                 absoluteLength++;
178             }
179         }
180         else
181         {
182             var absoluteLength = offset + relativeLength;
183             while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
184             {
185                 relativeLength++;
186                 absoluteLength++;
187             }
188         }
189         // copy array

```

```

187         var innerSequence = new ulong[relativeLength];
188         var maxLength = offset + relativeLength;
189         for (var i = offset; i < maxLength; i++)
190         {
191             innerSequence[i - offset] = array[i];
192         }
193         result.Add(innerSequence);
194         offset += relativeLength;
195     }
196     return result;
197 }
198 }
199 }

```

1.109 ./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Interfaces;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Unicode
8 {
9     public class UnicodeSequenceCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
10     ↪ ICriterionMatcher<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13         ↪ EqualityComparer<TLink>.Default;
14
15         private readonly TLink _unicodeSequenceMarker;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public UnicodeSequenceCriterionMatcher(ILinks<TLink> links, TLink unicodeSequenceMarker)
19         ↪ : base(links) => _unicodeSequenceMarker = unicodeSequenceMarker;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public bool IsMatched(TLink link) => _equalityComparer.Equals(_links.GetTarget(link),
23         ↪ _unicodeSequenceMarker);
24     }
25 }

```

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

```

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

```

```

34         var charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.
35             ↪ Convert).ToArray();
36         return new string(charArray);
37     }
38 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Interfaces;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Unicode
8 {
9     public class UnicodeSymbolCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
10         ↪ ICriterionMatcher<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↪ EqualityComparer<TLink>.Default;
14
15         private readonly TLink _unicodeSymbolMarker;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public UnicodeSymbolCriterionMatcher(ILinks<TLink> links, TLink unicodeSymbolMarker) :
19             ↪ base(links) => _unicodeSymbolMarker = unicodeSymbolMarker;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public bool IsMatched(TLink link) => _equalityComparer.Equals(_links.GetTarget(link),
23             ↪ _unicodeSymbolMarker);
24     }
25 }

```

1.112 ./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs

```

1 using System;
2 using System.Runtime.CompilerServices;
3 using Platform.Interfaces;
4 using Platform.Converters;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Unicode
9 {
10     public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
11         ↪ IConverter<TLink, char>
12     {
13         private static readonly UncheckedConverter<TLink, char> _addressToCharConverter =
14             ↪ UncheckedConverter<TLink, char>.Default;
15
16         private readonly IConverter<TLink> _numberToAddressConverter;
17         private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
21             ↪ numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
22             ↪ base(links)
23         {
24             _numberToAddressConverter = numberToAddressConverter;
25             _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
26         }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public char Convert(TLink source)
30         {
31             if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
32             {
33                 throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
34                     ↪ not a unicode symbol.");
35             }
36             return _addressToCharConverter.Convert(_numberToAddressConverter.Convert(_links.GetS
37                 ↪ ource(source)));
38         }
39     }
40 }

```

1.113 ./csharp/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

```

1.114 ./csharp/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     }
15 }

```

```

13     public int GetHashCode(ulong obj) => obj.GetHashCode();
14 }
15
16 private static bool Equals1<T>(T x, T y) => Equals(x, y);
17
18 private static bool Equals2<T>(T x, T y) => x.Equals(y);
19
20 private static bool Equals3(ulong x, ulong y) => x == y;
21
22 [Fact]
23 public static void EqualsPerformanceTest()
24 {
25     const int N = 1000000;
26
27     ulong x = 10;
28     ulong y = 500;
29
30     bool result = false;
31
32     var ts1 = Performance.Measure(() =>
33     {
34         for (int i = 0; i < N; i++)
35         {
36             result = Equals1(x, y);
37         }
38     });
39
40     var ts2 = Performance.Measure(() =>
41     {
42         for (int i = 0; i < N; i++)
43         {
44             result = Equals2(x, y);
45         }
46     });
47
48     var ts3 = Performance.Measure(() =>
49     {
50         for (int i = 0; i < N; i++)
51         {
52             result = Equals3(x, y);
53         }
54     });
55
56     var equalityComparer1 = EqualityComparer<ulong>.Default;
57
58     var ts4 = Performance.Measure(() =>
59     {
60         for (int i = 0; i < N; i++)
61         {
62             result = equalityComparer1.Equals(x, y);
63         }
64     });
65
66     var equalityComparer2 = new UInt64EqualityComparer();
67
68     var ts5 = Performance.Measure(() =>
69     {
70         for (int i = 0; i < N; i++)
71         {
72             result = equalityComparer2.Equals(x, y);
73         }
74     });
75
76     Func<ulong, ulong, bool> equalityComparer3 = equalityComparer2.Equals;
77
78     var ts6 = Performance.Measure(() =>
79     {
80         for (int i = 0; i < N; i++)
81         {
82             result = equalityComparer3(x, y);
83         }
84     });
85
86     var comparer = Comparer<ulong>.Default;
87
88     var ts7 = Performance.Measure(() =>
89     {
90         for (int i = 0; i < N; i++)
91         {
92

```

```

93         result = comparer.Compare(x, y) == 0;
94     }
95 });
96
97     Assert.True(ts2 < ts1);
98     Assert.True(ts3 < ts2);
99     Assert.True(ts5 < ts4);
100    Assert.True(ts5 < ts6);
101
102    Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {result}");
103 }
104 }
105 }

```

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

```

1  using System;
2  using Xunit;
3  using Platform.Reflection;
4  using Platform.Memory;
5  using Platform.Scopes;
6  using Platform.Data.Doublets.ResizableDirectMemory.Generic;
7
8  namespace Platform.Data.Doublets.Tests
9  {
10     public unsafe static class GenericLinksTests
11     {
12         [Fact]
13         public static void CRUDTest()
14         {
15             Using<byte>(links => links.TestCRUDOperations());
16             Using<ushort>(links => links.TestCRUDOperations());
17             Using<uint>(links => links.TestCRUDOperations());
18             Using<ulong>(links => links.TestCRUDOperations());
19         }
20
21         [Fact]
22         public static void RawNumbersCRUDTest()
23         {
24             Using<byte>(links => links.TestRawNumbersCRUDOperations());
25             Using<ushort>(links => links.TestRawNumbersCRUDOperations());
26             Using<uint>(links => links.TestRawNumbersCRUDOperations());
27             Using<ulong>(links => links.TestRawNumbersCRUDOperations());
28         }
29
30         [Fact]
31         public static void MultipleRandomCreationsAndDeletionsTest()
32         {
33             Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
34                 ↳ MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
35                 ↳ implementation of tree cuts out 5 bits from the address space.
36             Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
37                 ↳ stMultipleRandomCreationsAndDeletions(100));
38             Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
39                 ↳ MultipleRandomCreationsAndDeletions(100));
40             Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
41                 ↳ tMultipleRandomCreationsAndDeletions(100));
42         }
43
44         private static void Using<TLink>(Action<ILinks<TLink>> action)
45         {
46             using (var scope = new Scope<Types<HeapResizableDirectMemory,
47                 ↳ ResizableDirectMemoryLinks<TLink>>>())
48             {
49                 action(scope.Use<ILinks<TLink>>());
50             }
51         }
52     }
53 }

```

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

```

1  using Xunit;
2
3  namespace Platform.Data.Doublets.Tests
4  {
5     public static class LinksConstantsTests
6     {
7         [Fact]
8         public static void ExternalReferencesTest()

```

```

9      {
10         LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
            ↳ (long.MaxValue + 1UL, ulong.MaxValue));
11
12         //var minimum = new Hybrid<ulong>(0, isExternal: true);
13         var minimum = new Hybrid<ulong>(1, isExternal: true);
14         var maximum = new Hybrid<ulong>(long.MaxValue, isExternal: true);
15
16         Assert.True(constants.IsExternalReference(minimum));
17         Assert.True(constants.IsExternalReference(maximum));
18     }
19 }
20 }

```

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

```

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

```


Pulvinar mattis nunc sed blandit libero volutpat.
Cras fermentum odio eu feugiat pretium nibh ipsum.
In nulla posuere sollicitudin aliquam ultrices sagittis orci a.
Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.
A iaculis at erat pellentesque.
Morbi blandit cursus risus at ultrices mi tempus imperdiet nulla.
Eget lorem dolor sed viverra ipsum nunc.
Leo a diam sollicitudin tempor id eu.
Interdum consectetur libero id faucibus nisl tincidunt eget nullam non.";

[Fact]

```
public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
{
    using (var scope = new TempLinksTestScope(useSequences: false))
    {
        var links = scope.Links;
        var constants = links.Constants;

        links.UseUnicode();

        var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);

        var meaningRoot = links.CreatePoint();
        var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
        var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
        var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
            ↪ constants.Itself);

        var unaryNumberToAddressConverter = new
            ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
        var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
        var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
            ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
        var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
            ↪ frequencyPropertyMarker, frequencyMarker);
        var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
            ↪ frequencyPropertyOperator, frequencyIncrementer);
        var linkToItsFrequencyNumberConverter = new
            ↪ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
            ↪ unaryNumberToAddressConverter);
        var sequenceToItsLocalElementLevelsConverter = new
            ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
            ↪ linkToItsFrequencyNumberConverter);
        var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
            ↪ sequenceToItsLocalElementLevelsConverter);

        var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
            ↪ Walker = new LeveledSequenceWalker<ulong>(links) });

        ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
            ↪ index, optimalVariantConverter);
    }
}
```

[Fact]

```
public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
{
    using (var scope = new TempLinksTestScope(useSequences: false))
    {
        var links = scope.Links;

        links.UseUnicode();

        var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);

        var totalSequenceSymbolFrequencyCounter = new
            ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);

        var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
            ↪ totalSequenceSymbolFrequencyCounter);

        var index = new
            ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
        var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);

        var sequenceToItsLocalElementLevelsConverter = new
            ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
            ↪ linkToItsFrequencyNumberConverter);
```

```

128         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
129             ↪ sequenceToItsLocalElementLevelsConverter);
130
131         var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
132             ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
133
134         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
135             ↪ index, optimalVariantConverter);
136     }
137
138     private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
139         ↪ SequenceToItsLocalElementLevelsConverter<ulong>
140         ↪ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
141         ↪ OptimalVariantConverter<ulong> optimalVariantConverter)
142     {
143         index.Add(sequence);
144
145         var optimalVariant = optimalVariantConverter.Convert(sequence);
146
147         var readSequence1 = sequences.ToList(optimalVariant);
148
149         Assert.True(sequence.SequenceEqual(readSequence1));
150     }
151
152     [Fact]
153     public static void SavedSequencesOptimizationTest()
154     {
155         LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
156             ↪ (long.MaxValue + 1UL, ulong.MaxValue));
157
158         using (var memory = new HeapResizableDirectMemory())
159         using (var disposableLinks = new UInt64ResizableDirectMemoryLinks(memory,
160             ↪ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep, constants,
161             ↪ useAvlBasedIndex: false))
162         {
163             var links = new UInt64Links(disposableLinks);
164
165             var root = links.CreatePoint();
166
167             //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
168             var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
169
170             var unicodeSymbolMarker = links.GetOrCreate(root,
171                 ↪ addressToNumberConverter.Convert(1));
172             var unicodeSequenceMarker = links.GetOrCreate(root,
173                 ↪ addressToNumberConverter.Convert(2));
174
175             var totalSequenceSymbolFrequencyCounter = new
176                 ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
177             var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
178                 ↪ totalSequenceSymbolFrequencyCounter);
179             var index = new
180                 ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
181             var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
182                 ↪ ncyNumberConverter<ulong>(linkFrequenciesCache);
183             var sequenceToItsLocalElementLevelsConverter = new
184                 ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
185                 ↪ linkToItsFrequencyNumberConverter);
186             var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
187                 ↪ sequenceToItsLocalElementLevelsConverter);
188
189             var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
190                 ↪ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
191
192             var unicodeSequencesOptions = new SequencesOptions<ulong>()
193             {
194                 UseSequenceMarker = true,
195                 SequenceMarkerLink = unicodeSequenceMarker,
196                 UseIndex = true,
197                 Index = index,
198                 LinksToSequenceConverter = optimalVariantConverter,
199                 Walker = walker,
200                 UseGarbageCollection = true
201             };
202
203             var unicodeSequences = new Sequences.Sequences(new
204                 ↪ SynchronizedLinks<ulong>(links), unicodeSequencesOptions);

```

```

186
187 // Create some sequences
188 var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
    ↳ StringSplitOptions.RemoveEmptyEntries);
189 var arrays = strings.Select(x => x.Select(y =>
    ↳ addressToNumberConverter.Convert(y)).ToArray()).ToArray();
190 for (int i = 0; i < arrays.Length; i++)
191 {
192     unicodeSequences.Create(arrays[i].ShiftRight());
193 }
194
195 var linksCountAfterCreation = links.Count();
196
197 // get list of sequences links
198 // for each sequence link
199 //     create new sequence version
200 //     if new sequence is not the same as sequence link
201 //         delete sequence link
202 //         collect garbadge
203 unicodeSequences.CompactAll();
204
205 var linksCountAfterCompactification = links.Count();
206
207 Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
208 }
209 }
210 }
211 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Diagnostics;
4 using System.Linq;
5 using Xunit;
6 using Platform.Data.Sequences;
7 using Platform.Data.Doublets.Sequences.Converters;
8 using Platform.Data.Doublets.Sequences.Walkers;
9 using Platform.Data.Doublets.Sequences;
10
11 namespace Platform.Data.Doublets.Tests
12 {
13     public static class ReadSequenceTests
14     {
15         [Fact]
16         public static void ReadSequenceTest()
17         {
18             const long sequenceLength = 2000;
19
20             using (var scope = new TempLinksTestScope(useSequences: false))
21             {
22                 var links = scope.Links;
23                 var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong> {
24                     ↳ Walker = new LeveledSequenceWalker<ulong>(links) });
25
26                 var sequence = new ulong[sequenceLength];
27                 for (var i = 0; i < sequenceLength; i++)
28                 {
29                     sequence[i] = links.Create();
30                 }
31
32                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
33
34                 var sw1 = Stopwatch.StartNew();
35                 var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
36
37                 var sw2 = Stopwatch.StartNew();
38                 var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();
39
40                 var sw3 = Stopwatch.StartNew();
41                 var readSequence2 = new List<ulong>();
42                 SequenceWalker.WalkRight(balancedVariant,
43                                         links.GetSource,
44                                         links.GetTarget,
45                                         links.IsPartialPoint,
46                                         readSequence2.Add);
47
48                 sw3.Stop();
49
50                 Assert.True(sequence.SequenceEqual(readSequence1));
51             }
52         }
53     }
54 }

```

```

50         Assert.True(sequence.SequenceEqual(readSequence2));
51
52         // Assert.True(sw2.Elapsed < sw3.Elapsed);
53
54         Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
55         ↳ {sw2.Elapsed}");
56
57         for (var i = 0; i < sequenceLength; i++)
58         {
59             links.Delete(sequence[i]);
60         }
61     }
62 }
63 }

```

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

```

1  using System.IO;
2  using Xunit;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using Platform.Data.Doublets.ResizableDirectMemory.Specific;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public static class ResizableDirectMemoryLinksTests
10     {
11         private static readonly LinksConstants<ulong> _constants =
12         ↳ Default<LinksConstants<ulong>>.Instance;
13
14         [Fact]
15         public static void BasicFileMappedMemoryTest()
16         {
17             var tempFilename = Path.GetTempFileName();
18             using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(tempFilename))
19             {
20                 memoryAdapter.TestBasicMemoryOperations();
21             }
22             File.Delete(tempFilename);
23
24             [Fact]
25             public static void BasicHeapMemoryTest()
26             {
27                 using (var memory = new
28                 ↳ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
29                 using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
30                 ↳ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
31                 {
32                     memoryAdapter.TestBasicMemoryOperations();
33                 }
34
35                 private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
36                 {
37                     var link = memoryAdapter.Create();
38                     memoryAdapter.Delete(link);
39                 }
40
41                 [Fact]
42                 public static void NonexistentReferencesHeapMemoryTest()
43                 {
44                     using (var memory = new
45                     ↳ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
46                     using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
47                     ↳ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
48                     {
49                         memoryAdapter.TestNonexistentReferences();
50                     }
51
52                     private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
53                     {
54                         var link = memoryAdapter.Create();
55                         memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
56                         var resultLink = _constants.Null;
57                         memoryAdapter.Each(foundLink =>
58                         {
59                             resultLink = foundLink[_constants.IndexPart];

```

```

58         return _constants.Break;
59     }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
60     Assert.True(resultLink == link);
61     Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
62     memoryAdapter.Delete(link);
63 }
64 }
65 }

```

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

```

1  using Xunit;
2  using Platform.Scopes;
3  using Platform.Memory;
4  using Platform.Data.Doublets.Decorators;
5  using Platform.Reflection;
6  using Platform.Data.Doublets.ResizableDirectMemory.Generic;
7  using Platform.Data.Doublets.ResizableDirectMemory.Specific;
8
9  namespace Platform.Data.Doublets.Tests
10 {
11     public static class ScopeTests
12     {
13         [Fact]
14         public static void SingleDependencyTest()
15         {
16             using (var scope = new Scope())
17             {
18                 scope.IncludeAssemblyOf<IMemory>();
19                 var instance = scope.Use<IDirectMemory>();
20                 Assert.IsType<HeapResizableDirectMemory>(instance);
21             }
22         }
23
24         [Fact]
25         public static void CascadeDependencyTest()
26         {
27             using (var scope = new Scope())
28             {
29                 scope.Include<TemporaryFileMappedResizableDirectMemory>();
30                 scope.Include<UInt64ResizableDirectMemoryLinks>();
31                 var instance = scope.Use<ILinks<ulong>>();
32                 Assert.IsType<UInt64ResizableDirectMemoryLinks>(instance);
33             }
34         }
35
36         [Fact]
37         public static void FullAutoResolutionTest()
38         {
39             using (var scope = new Scope(autoInclude: true, autoExplore: true))
40             {
41                 var instance = scope.Use<UInt64Links>();
42                 Assert.IsType<UInt64Links>(instance);
43             }
44         }
45
46         [Fact]
47         public static void TypeParametersTest()
48         {
49             using (var scope = new Scope<Types<HeapResizableDirectMemory,
50 ↵ ResizableDirectMemoryLinks<ulong>>>())
51             {
52                 var links = scope.Use<ILinks<ulong>>();
53                 Assert.IsType<ResizableDirectMemoryLinks<ulong>>(links);
54             }
55         }
56     }
57 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Linq;
5  using Xunit;
6  using Platform.Collections;
7  using Platform.Collections.Arrays;
8  using Platform.Random;
9  using Platform.IO;
10 using Platform.Singletons;

```

```

11 using Platform.Data.Doublets.Sequences;
12 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
13 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
14 using Platform.Data.Doublets.Sequences.Converters;
15 using Platform.Data.Doublets.Unicode;
16
17 namespace Platform.Data.Doublets.Tests
18 {
19     public static class SequencesTests
20     {
21         private static readonly LinksConstants<ulong> _constants =
22             ↳ Default<LinksConstants<ulong>>.Instance;
23
24         static SequencesTests()
25         {
26             // Trigger static constructor to not mess with performance measurements
27             _ = BitString.GetBitMaskFromIndex(1);
28         }
29
30         [Fact]
31         public static void CreateAllVariantsTest()
32         {
33             const long sequenceLength = 8;
34
35             using (var scope = new TempLinksTestScope(useSequences: true))
36             {
37                 var links = scope.Links;
38                 var sequences = scope.Sequences;
39
40                 var sequence = new ulong[sequenceLength];
41                 for (var i = 0; i < sequenceLength; i++)
42                 {
43                     sequence[i] = links.Create();
44                 }
45
46                 var sw1 = Stopwatch.StartNew();
47                 var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
48
49                 var sw2 = Stopwatch.StartNew();
50                 var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
51
52                 Assert.True(results1.Count > results2.Length);
53                 Assert.True(sw1.Elapsed > sw2.Elapsed);
54
55                 for (var i = 0; i < sequenceLength; i++)
56                 {
57                     links.Delete(sequence[i]);
58                 }
59
60                 Assert.True(links.Count() == 0);
61             }
62         }
63
64         //[Fact]
65         //public void CUDTest()
66         //{
67             // var tempFilename = Path.GetTempFileName();
68
69             // const long sequenceLength = 8;
70
71             // const ulong itself = LinksConstants.Itself;
72
73             // using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
74             // ↳ DefaultLinksSizeStep))
75             // using (var links = new Links(memoryAdapter))
76             // {
77                 // var sequence = new ulong[sequenceLength];
78                 // for (var i = 0; i < sequenceLength; i++)
79                 //     sequence[i] = links.Create(itself, itself);
80
81                 // SequencesOptions o = new SequencesOptions();
82
83                 // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
84                 // o.
85
86                 // var sequences = new Sequences(links);
87
88                 // var sw1 = Stopwatch.StartNew();
89                 // var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
90             }
91         }
92     }
93 }

```

```

89 //         var sw2 = Stopwatch.StartNew();
90 //         var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
91
92 //         Assert.True(results1.Count > results2.Length);
93 //         Assert.True(sw1.Elapsed > sw2.Elapsed);
94
95 //         for (var i = 0; i < sequenceLength; i++)
96 //             links.Delete(sequence[i]);
97 //     }
98
99 //     File.Delete(tempFilename);
100 // }
101
102 [Fact]
103 public static void AllVariantsSearchTest()
104 {
105     const long sequenceLength = 8;
106
107     using (var scope = new TempLinksTestScope(useSequences: true))
108     {
109         var links = scope.Links;
110         var sequences = scope.Sequences;
111
112         var sequence = new ulong[sequenceLength];
113         for (var i = 0; i < sequenceLength; i++)
114         {
115             sequence[i] = links.Create();
116         }
117
118         var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
119
120         //for (int i = 0; i < createResults.Length; i++)
121         //    sequences.Create(createResults[i]);
122
123         var sw0 = Stopwatch.StartNew();
124         var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
125
126         var sw1 = Stopwatch.StartNew();
127         var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
128
129         var sw2 = Stopwatch.StartNew();
130         var searchResults2 = sequences.Each1(sequence); sw2.Stop();
131
132         var sw3 = Stopwatch.StartNew();
133         var searchResults3 = sequences.Each(sequence.ShiftRight()); sw3.Stop();
134
135         var intersection0 = createResults.Intersect(searchResults0).ToList();
136         Assert.True(intersection0.Count == searchResults0.Count);
137         Assert.True(intersection0.Count == createResults.Length);
138
139         var intersection1 = createResults.Intersect(searchResults1).ToList();
140         Assert.True(intersection1.Count == searchResults1.Count);
141         Assert.True(intersection1.Count == createResults.Length);
142
143         var intersection2 = createResults.Intersect(searchResults2).ToList();
144         Assert.True(intersection2.Count == searchResults2.Count);
145         Assert.True(intersection2.Count == createResults.Length);
146
147         var intersection3 = createResults.Intersect(searchResults3).ToList();
148         Assert.True(intersection3.Count == searchResults3.Count);
149         Assert.True(intersection3.Count == createResults.Length);
150
151         for (var i = 0; i < sequenceLength; i++)
152         {
153             links.Delete(sequence[i]);
154         }
155     }
156 }
157
158 [Fact]
159 public static void BalancedVariantSearchTest()
160 {
161     const long sequenceLength = 200;
162
163     using (var scope = new TempLinksTestScope(useSequences: true))
164     {
165         var links = scope.Links;
166         var sequences = scope.Sequences;
167
168         var sequence = new ulong[sequenceLength];

```

```

169     for (var i = 0; i < sequenceLength; i++)
170     {
171         sequence[i] = links.Create();
172     }
173
174     var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
175
176     var sw1 = Stopwatch.StartNew();
177     var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
178
179     var sw2 = Stopwatch.StartNew();
180     var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
181
182     var sw3 = Stopwatch.StartNew();
183     var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
184
185     // На количестве в 200 элементов это будет занимать вечность
186     //var sw4 = Stopwatch.StartNew();
187     //var searchResults4 = sequences.Each(sequence); sw4.Stop();
188
189     Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
190
191     Assert.True(searchResults3.Count == 1 && balancedVariant ==
192         ↪ searchResults3.First());
193
194     //Assert.True(sw1.Elapsed < sw2.Elapsed);
195
196     for (var i = 0; i < sequenceLength; i++)
197     {
198         links.Delete(sequence[i]);
199     }
200 }
201
202 [Fact]
203 public static void AllPartialVariantsSearchTest()
204 {
205     const long sequenceLength = 8;
206
207     using (var scope = new TempLinksTestScope(useSequences: true))
208     {
209         var links = scope.Links;
210         var sequences = scope.Sequences;
211
212         var sequence = new ulong[sequenceLength];
213         for (var i = 0; i < sequenceLength; i++)
214         {
215             sequence[i] = links.Create();
216         }
217
218         var createResults = sequences.CreateAllVariants2(sequence);
219
220         //var createResultsStrings = createResults.Select(x => x + ": " +
221             ↪ sequences.FormatSequence(x)).ToList();
222         //Global.Trash = createResultsStrings;
223
224         var partialSequence = new ulong[sequenceLength - 2];
225
226         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
227
228         var sw1 = Stopwatch.StartNew();
229         var searchResults1 =
230             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
231
232         var sw2 = Stopwatch.StartNew();
233         var searchResults2 =
234             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
235
236         //var sw3 = Stopwatch.StartNew();
237         //var searchResults3 =
238             ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
239
240         var sw4 = Stopwatch.StartNew();
241         var searchResults4 =
242             ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
243
244         //Global.Trash = searchResults3;
245
246         //var searchResults1Strings = searchResults1.Select(x => x + ": " +
247             ↪ sequences.FormatSequence(x)).ToList();

```



```

242 //Global.Trash = searchResults1Strings;
243
244 var intersection1 = createResults.Intersect(searchResults1).ToList();
245 Assert.True(intersection1.Count == createResults.Length);
246
247 var intersection2 = createResults.Intersect(searchResults2).ToList();
248 Assert.True(intersection2.Count == createResults.Length);
249
250 var intersection4 = createResults.Intersect(searchResults4).ToList();
251 Assert.True(intersection4.Count == createResults.Length);
252
253 for (var i = 0; i < sequenceLength; i++)
254 {
255     links.Delete(sequence[i]);
256 }
257 }
258 }
259
260 [Fact]
261 public static void BalancedPartialVariantsSearchTest()
262 {
263     const long sequenceLength = 200;
264
265     using (var scope = new TempLinksTestScope(useSequences: true))
266     {
267         var links = scope.Links;
268         var sequences = scope.Sequences;
269
270         var sequence = new ulong[sequenceLength];
271         for (var i = 0; i < sequenceLength; i++)
272         {
273             sequence[i] = links.Create();
274         }
275
276         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
277
278         var balancedVariant = balancedVariantConverter.Convert(sequence);
279
280         var partialSequence = new ulong[sequenceLength - 2];
281
282         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
283
284         var sw1 = Stopwatch.StartNew();
285         var searchResults1 =
286             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
287
288         var sw2 = Stopwatch.StartNew();
289         var searchResults2 =
290             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
291
292         Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
293
294         Assert.True(searchResults2.Count == 1 && balancedVariant ==
295             ↪ searchResults2.First());
296
297         for (var i = 0; i < sequenceLength; i++)
298         {
299             links.Delete(sequence[i]);
300         }
301     }
302 }
303
304 [Fact(Skip = "Correct implementation is pending")]
305 public static void PatternMatchTest()
306 {
307     var zeroOrMany = Sequences.Sequences.ZeroOrMany;
308
309     using (var scope = new TempLinksTestScope(useSequences: true))
310     {
311         var links = scope.Links;
312         var sequences = scope.Sequences;
313
314         var e1 = links.Create();
315         var e2 = links.Create();
316
317         var sequence = new[]
318         {
319             e1, e2, e1, e2 // mama / papa
320         };
321     }
322 }

```

```

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

412
413 [![белая связь, чёрная направленная
→ связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png "белая
→ связь, чёрная направленная
→ связь")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png)

414
415 Допустим у нас есть смысл "связать" и смысл "направления", много ли это нам даёт? Много ли
→ вариантов интерпретации? А что если уточнить, каким именно образом выполнена связь? Что если
→ можно задать ей чёткий, конкретный смысл? Что это будет? Тип? Глагол? Связка? Действие?
→ Трансформация? Переход из состояния в состояние? Или всё это и есть объект, суть которого в
→ его конечном состоянии, если конечно конец определён направлением?

416
417 [![белая обычная и направленная связи, чёрная типизированная
→ связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png "белая
→ обычная и направленная связи, чёрная типизированная
→ связь")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png)

418
419 А что если всё это время, мы смотрели на суть как бы снаружи? Можно ли взглянуть на это изнутри?
→ Что будет внутри объектов? Объекты ли это? Или это связи? Может ли эта структура описать
→ сама себя? Но что тогда получится, разве это не рекурсия? Может это фрактал?

420

```

421 [![белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная
    ↳ связь с рекурсивной внутренней
    ↳ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png
    ↳ ""белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с рекурсивной внутренней структурой"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png)
422
423 На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
    ↳ рекурсии или фрактала?
424
425 [![белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с двойной рекурсивной внутренней
    ↳ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png
    ↳ ""белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с двойной рекурсивной внутренней структурой"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)
426
427 Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
    ↳ Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
428
429 [![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности,
    ↳ чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://
    ↳ raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png ""белая обычная и
    ↳ направленная связи со структурой из 8 цветных элементов последовательности, чёрная
    ↳ типизированная связь со структурой из 8 цветных элементов последовательности"")](https://raw
    ↳ .githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)
430
431 ...
432
433 [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-anim
    ↳ ation-500.gif
    ↳ ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
    ↳ -animation-500.gif)";
434
435     private static readonly string _exampleLoremIpsumText =
436         @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
            ↳ incididunt ut labore et dolore magna aliqua.
437 Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
    ↳ consequat.";
438
439 [Fact]
440 public static void CompressionTest()
441 {
442     using (var scope = new TempLinksTestScope(useSequences: true))
443     {
444         var links = scope.Links;
445         var sequences = scope.Sequences;
446
447         var e1 = links.Create();
448         var e2 = links.Create();
449
450         var sequence = new[]
451         {
452             e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
453         };
454
455         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
456         var totalSequenceSymbolFrequencyCounter = new
            ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
457         var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
            ↳ totalSequenceSymbolFrequencyCounter);
458         var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
            ↳ balancedVariantConverter, doubletFrequenciesCache);
459
460         var compressedVariant = compressingConverter.Convert(sequence);
461
462         // 1: [1]          (1->1) point
463         // 2: [2]          (2->2) point
464         // 3: [1,2]        (1->2) doublet
465         // 4: [1,2,1,2]    (3->3) doublet
466
467         Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
468         Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
469         Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
470         Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
471
472         var source = _constants.SourcePart;
473         var target = _constants.TargetPart;
474

```

```

475 Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
476 Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
477 Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
478 Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
479
480 // 4 - length of sequence
481 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
482   ↳ == sequence[0]);
483 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
484   ↳ == sequence[1]);
485 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
486   ↳ == sequence[2]);
487 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
488   ↳ == sequence[3]);
489
490 }
491
492 [Fact]
493 public static void CompressionEfficiencyTest()
494 {
495     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
496   ↳ StringSplitOptions.RemoveEmptyEntries);
497     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
498     var totalCharacters = arrays.Select(x => x.Length).Sum();
499
500     using (var scope1 = new TempLinksTestScope(useSequences: true))
501     using (var scope2 = new TempLinksTestScope(useSequences: true))
502     using (var scope3 = new TempLinksTestScope(useSequences: true))
503     {
504         scope1.Links.Unsync.UseUnicode();
505         scope2.Links.Unsync.UseUnicode();
506         scope3.Links.Unsync.UseUnicode();
507
508         var balancedVariantConverter1 = new
509   ↳ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
510         var totalSequenceSymbolFrequencyCounter = new
511   ↳ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
512         var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
513   ↳ totalSequenceSymbolFrequencyCounter);
514         var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
515   ↳ balancedVariantConverter1, linkFrequenciesCache1,
516   ↳ doInitialFrequenciesIncrement: false);
517
518         //var compressor2 = scope2.Sequences;
519         var compressor3 = scope3.Sequences;
520
521         var constants = Default<LinksConstants<ulong>>.Instance;
522
523         var sequences = compressor3;
524         //var meaningRoot = links.CreatePoint();
525         //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
526         //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
527         //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
528   ↳ constants.Itself);
529
530         //var unaryNumberToAddressConverter = new
531   ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
532         //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
533   ↳ unaryOne);
534         //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
535   ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
536         //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
537   ↳ frequencyPropertyMarker, frequencyMarker);
538         //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
539   ↳ frequencyPropertyOperator, frequencyIncrementer);
540         //var linkToItsFrequencyNumberConverter = new
541   ↳ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
542   ↳ unaryNumberToAddressConverter);
543
544         var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
545   ↳ totalSequenceSymbolFrequencyCounter);
546
547         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque_
548   ↳ ncyNumberConverter<ulong>(linkFrequenciesCache3);

```

```

530     var sequenceToItsLocalElementLevelsConverter = new
        ↳ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
        ↳ linkToItsFrequencyNumberConverter);
531     var optimalVariantConverter = new
        ↳ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
        ↳ sequenceToItsLocalElementLevelsConverter);

532
533     var compressed1 = new ulong[arrays.Length];
534     var compressed2 = new ulong[arrays.Length];
535     var compressed3 = new ulong[arrays.Length];
536
537     var START = 0;
538     var END = arrays.Length;
539
540     //for (int i = START; i < END; i++)
541     //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
542
543     var initialCount1 = scope2.Links.Unsync.Count();
544
545     var sw1 = Stopwatch.StartNew();
546
547     for (int i = START; i < END; i++)
548     {
549         linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
550         compressed1[i] = compressor1.Convert(arrays[i]);
551     }
552
553     var elapsed1 = sw1.Elapsed;
554
555     var balancedVariantConverter2 = new
        ↳ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
556
557     var initialCount2 = scope2.Links.Unsync.Count();
558
559     var sw2 = Stopwatch.StartNew();
560
561     for (int i = START; i < END; i++)
562     {
563         compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
564     }
565
566     var elapsed2 = sw2.Elapsed;
567
568     for (int i = START; i < END; i++)
569     {
570         linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
571     }
572
573     var initialCount3 = scope3.Links.Unsync.Count();
574
575     var sw3 = Stopwatch.StartNew();
576
577     for (int i = START; i < END; i++)
578     {
579         //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
580         compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
581     }
582
583     var elapsed3 = sw3.Elapsed;
584
585     Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
        ↳ Optimal variant: {elapsed3}");
586
587     // Assert.True(elapsed1 > elapsed2);
588
589     // Checks
590     for (int i = START; i < END; i++)
591     {
592         var sequence1 = compressed1[i];
593         var sequence2 = compressed2[i];
594         var sequence3 = compressed3[i];
595
596         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
            ↳ scope1.Links.Unsync);
597
598         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
            ↳ scope2.Links.Unsync);
599
600         var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
            ↳ scope3.Links.Unsync);

```

```

601
602     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
        ↳ link.IsPartialPoint());
603     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
        ↳ link.IsPartialPoint());
604     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
        ↳ link.IsPartialPoint());
605
606     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
        ↳ arrays[i].Length > 3)
607     //    Assert.False(structure1 == structure2);
608     //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
        ↳ arrays[i].Length > 3)
609     //    Assert.False(structure3 == structure2);
610
611     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
612     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
613 }
614
615 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
        ↳ totalCharacters);
616 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
        ↳ totalCharacters);
617 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
        ↳ totalCharacters);
618
619 Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
        ↳ totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
        ↳ totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
        ↳ totalCharacters}");
620
621 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
        ↳ scope2.Links.Unsync.Count() - initialCount2);
622 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
        ↳ scope2.Links.Unsync.Count() - initialCount2);
623
624 var duplicateProvider1 = new
        ↳ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
625 var duplicateProvider2 = new
        ↳ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
626 var duplicateProvider3 = new
        ↳ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
627
628 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
629 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
630 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
631
632 var duplicates1 = duplicateCounter1.Count();
633
634 ConsoleHelpers.Debug("-----");
635
636 var duplicates2 = duplicateCounter2.Count();
637
638 ConsoleHelpers.Debug("-----");
639
640 var duplicates3 = duplicateCounter3.Count();
641
642 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
643
644 linkFrequenciesCache1.ValidateFrequencies();
645 linkFrequenciesCache3.ValidateFrequencies();
646 }
647 }
648
649 [Fact]
650 public static void CompressionStabilityTest()
651 {
652     // TODO: Fix bug (do a separate test)
653     //const ulong minNumbers = 0;
654     //const ulong maxNumbers = 1000;
655
656     const ulong minNumbers = 10000;
657     const ulong maxNumbers = 12500;
658
659     var strings = new List<string>();
660
661     for (ulong i = minNumbers; i < maxNumbers; i++)
662     {

```

```

663         strings.Add(i.ToString());
664     }
665
666     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
667     var totalCharacters = arrays.Select(x => x.Length).Sum();
668
669     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
        ↳ SequencesOptions<ulong> { UseCompression = true,
        ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
        using (var scope2 = new TempLinksTestScope(useSequences: true))
        {
            scope1.Links.UseUnicode();
            scope2.Links.UseUnicode();
            //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
            var compressor1 = scope1.Sequences;
            var compressor2 = scope2.Sequences;
            var compressed1 = new ulong[arrays.Length];
            var compressed2 = new ulong[arrays.Length];
            var sw1 = Stopwatch.StartNew();
            var START = 0;
            var END = arrays.Length;
            // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
            // Stability issue starts at 10001 or 11000
            //for (int i = START; i < END; i++)
            //{
            //    var first = compressor1.Compress(arrays[i]);
            //    var second = compressor1.Compress(arrays[i]);
            //    if (first == second)
            //        compressed1[i] = first;
            //    else
            //    {
            //        // TODO: Find a solution for this case
            //    }
            //}
            for (int i = START; i < END; i++)
            {
                var first = compressor1.Create(arrays[i].ShiftRight());
                var second = compressor1.Create(arrays[i].ShiftRight());
                if (first == second)
                {
                    compressed1[i] = first;
                }
                else
                {
                    // TODO: Find a solution for this case
                }
            }
            var elapsed1 = sw1.Elapsed;
            var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
            var sw2 = Stopwatch.StartNew();
            for (int i = START; i < END; i++)
            {
                var first = balancedVariantConverter.Convert(arrays[i]);
                var second = balancedVariantConverter.Convert(arrays[i]);
                if (first == second)
                {
                    compressed2[i] = first;
                }
            }
            var elapsed2 = sw2.Elapsed;
            Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
        ↳ {elapsed2}");
            Assert.True(elapsed1 > elapsed2);

```



```

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
780 [Fact]
781 public static void RandomNumbersCompressionQualityTest()
782 {
783     const ulong N = 500;
784
785     //const ulong minNumbers = 10000;
786     //const ulong maxNumbers = 20000;
787
788     //var strings = new List<string>();
789
790     //for (ulong i = 0; i < N; i++)
791     //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
792         ↪ maxNumbers).ToString());
793
794     var strings = new List<string>();
795
796     for (ulong i = 0; i < N; i++)
797     {
798         strings.Add(RandomHelpers.Default.NextUInt64().ToString());
799     }
800
801     strings = strings.Distinct().ToList();
802
803     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
804     var totalCharacters = arrays.Select(x => x.Length).Sum();
805
806     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
807         ↪ SequencesOptions<ulong> { UseCompression = true,
808         ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
809     using (var scope2 = new TempLinksTestScope(useSequences: true))
810     {
811         scope1.Links.UseUnicode();
812         scope2.Links.UseUnicode();
813
814         var compressor1 = scope1.Sequences;
815         var compressor2 = scope2.Sequences;
816
817         var compressed1 = new ulong[arrays.Length];
818         var compressed2 = new ulong[arrays.Length];

```

```

809
810     var sw1 = Stopwatch.StartNew();
811
812     var START = 0;
813     var END = arrays.Length;
814
815     for (int i = START; i < END; i++)
816     {
817         compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
818     }
819
820     var elapsed1 = sw1.Elapsed;
821
822     var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
823
824     var sw2 = Stopwatch.StartNew();
825
826     for (int i = START; i < END; i++)
827     {
828         compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
829     }
830
831     var elapsed2 = sw2.Elapsed;
832
833     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
834         ↪ {elapsed2}");
835
836     Assert.True(elapsed1 > elapsed2);
837
838     // Checks
839     for (int i = START; i < END; i++)
840     {
841         var sequence1 = compressed1[i];
842         var sequence2 = compressed2[i];
843
844         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
845         {
846             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
847                 ↪ scope1.Links);
848
849             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
850                 ↪ scope2.Links);
851
852             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
853         }
854     }
855
856     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
857     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
858
859     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
860         ↪ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
861         ↪ totalCharacters}}");
862
863     // Can be worse than balanced variant
864     //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
865
866     //compressor1.ValidateFrequencies();
867 }
868
869 [Fact]
870 public static void AllTreeBreakDownAtSequencesCreationBugTest()
871 {
872     // Made out of AllPossibleConnectionsTest test.
873
874     //const long sequenceLength = 5; //100% bug
875     const long sequenceLength = 4; //100% bug
876     //const long sequenceLength = 3; //100% _no_bug (ok)
877
878     using (var scope = new TempLinksTestScope(useSequences: true))
879     {
880         var links = scope.Links;
881         var sequences = scope.Sequences;
882
883         var sequence = new ulong[sequenceLength];
884         for (var i = 0; i < sequenceLength; i++)
885         {
886             sequence[i] = links.Create();
887         }
888     }
889 }

```

```

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

```

```

963     var sequences = scope.Sequences;
964
965     var sequence = new ulong[sequenceLength];
966     for (var i = 0; i < sequenceLength; i++)
967     {
968         sequence[i] = links.Create();
969     }
970
971     var createResults = sequences.CreateAllVariants2(sequence);
972
973     //var reverseResults =
974     ↪ sequences.CreateAllVariants2(sequence.Reverse().ToArray());
975
976     for (var i = 0; i < 1; i++)
977     {
978         var linksTotalUsages1 = new ulong[links.Count() + 1];
979
980         sequences.CalculateAllUsages(linksTotalUsages1);
981
982         var linksTotalUsages2 = new ulong[links.Count() + 1];
983
984         sequences.CalculateAllUsages2(linksTotalUsages2);
985
986         var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
987         Assert.True(intersection1.Count == linksTotalUsages2.Length);
988     }
989
990     for (var i = 0; i < sequenceLength; i++)
991     {
992         links.Delete(sequence[i]);
993     }
994 }
995 }
996 }

```

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

```

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

```

```

38     {
39         using (var dataMemory = new HeapResizableDirectMemory())
40         using (var indexMemory = new HeapResizableDirectMemory())
41         using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory))
42         {
43             action(memory);
44         }
45     }
46 }
47 }

```

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

```

1  using System.IO;
2  using Platform.Disposables;
3  using Platform.Data.Doublets.Sequences;
4  using Platform.Data.Doublets.Decorators;
5  using Platform.Data.Doublets.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);
29             MemoryAdapter = useLog ? (ILinks<ulong>)new
30                 ↪ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
31                 ↪ coreMemoryAdapter;
32             Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
33             if (useSequences)
34             {
35                 Sequences = new Sequences.Sequences(Links, sequencesOptions);
36             }
37         }
38
39         protected override void Dispose(bool manual, bool wasDisposed)
40         {
41             if (!wasDisposed)
42             {
43                 Links.Unsync.DisposeIfPossible();
44                 if (_deleteFiles)
45                 {
46                     DeleteFiles();
47                 }
48             }
49         }
50
51         public void DeleteFiles()
52         {
53             File.Delete(TempFilename);
54             File.Delete(TempTransactionLogFilename);
55         }
56     }
57 }

```

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

```

1  using System.Collections.Generic;
2  using Xunit;
3  using Platform.Ranges;
4  using Platform.Numbers;
5  using Platform.Random;
6  using Platform.Setters;
7  using Platform.Converters;
8

```

```

9 namespace Platform.Data.Doublets.Tests
10 {
11     public static class TestExtensions
12     {
13         public static void TestCRUDOperations<T>(this ILinks<T> links)
14         {
15             var constants = links.Constants;
16
17             var equalityComparer = EqualityComparer<T>.Default;
18
19             var zero = default(T);
20             var one = Arithmetic.Increment(zero);
21
22             // Create Link
23             Assert.True(equalityComparer.Equals(links.Count(), zero));
24
25             var setter = new Setter<T>(constants.Null);
26             links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
27
28             Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
29
30             var linkAddress = links.Create();
31
32             var link = new Link<T>(links.GetLink(linkAddress));
33
34             Assert.True(link.Count == 3);
35             Assert.True(equalityComparer.Equals(link.Index, linkAddress));
36             Assert.True(equalityComparer.Equals(link.Source, constants.Null));
37             Assert.True(equalityComparer.Equals(link.Target, constants.Null));
38
39             Assert.True(equalityComparer.Equals(links.Count(), one));
40
41             // Get first link
42             setter = new Setter<T>(constants.Null);
43             links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
44
45             Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
46
47             // Update link to reference itself
48             links.Update(linkAddress, linkAddress, linkAddress);
49
50             link = new Link<T>(links.GetLink(linkAddress));
51
52             Assert.True(equalityComparer.Equals(link.Source, linkAddress));
53             Assert.True(equalityComparer.Equals(link.Target, linkAddress));
54
55             // Update link to reference null (prepare for delete)
56             var updated = links.Update(linkAddress, constants.Null, constants.Null);
57
58             Assert.True(equalityComparer.Equals(updated, linkAddress));
59
60             link = new Link<T>(links.GetLink(linkAddress));
61
62             Assert.True(equalityComparer.Equals(link.Source, constants.Null));
63             Assert.True(equalityComparer.Equals(link.Target, constants.Null));
64
65             // Delete link
66             links.Delete(linkAddress);
67
68             Assert.True(equalityComparer.Equals(links.Count(), zero));
69
70             setter = new Setter<T>(constants.Null);
71             links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
72
73             Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
74         }
75
76         public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
77         {
78             // Constants
79             var constants = links.Constants;
80             var equalityComparer = EqualityComparer<T>.Default;
81
82             var zero = default(T);
83             var one = Arithmetic.Increment(zero);
84             var two = Arithmetic.Increment(one);
85
86             var h106E = new Hybrid<T>(106L, isExternal: true);
87             var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
88             var h108E = new Hybrid<T>(-108L);

```

```

89
90     Assert.Equal(106L, h106E.AbsoluteValue);
91     Assert.Equal(107L, h107E.AbsoluteValue);
92     Assert.Equal(108L, h108E.AbsoluteValue);
93
94     // Create Link (External -> External)
95     var linkAddress1 = links.Create();
96
97     links.Update(linkAddress1, h106E, h108E);
98
99     var link1 = new Link<T>(links.GetLink(linkAddress1));
100
101     Assert.True(equalityComparer.Equals(link1.Source, h106E));
102     Assert.True(equalityComparer.Equals(link1.Target, h108E));
103
104     // Create Link (Internal -> External)
105     var linkAddress2 = links.Create();
106
107     links.Update(linkAddress2, linkAddress1, h108E);
108
109     var link2 = new Link<T>(links.GetLink(linkAddress2));
110
111     Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
112     Assert.True(equalityComparer.Equals(link2.Target, h108E));
113
114     // Create Link (Internal -> Internal)
115     var linkAddress3 = links.Create();
116
117     links.Update(linkAddress3, linkAddress1, linkAddress2);
118
119     var link3 = new Link<T>(links.GetLink(linkAddress3));
120
121     Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
122     Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
123
124     // Search for created link
125     var setter1 = new Setter<T>(constants.Null);
126     links.Each(h106E, h108E, setter1.SetAndReturnFalse);
127
128     Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
129
130     // Search for nonexistent link
131     var setter2 = new Setter<T>(constants.Null);
132     links.Each(h106E, h107E, setter2.SetAndReturnFalse);
133
134     Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
135
136     // Update link to reference null (prepare for delete)
137     var updated = links.Update(linkAddress3, constants.Null, constants.Null);
138
139     Assert.True(equalityComparer.Equals(updated, linkAddress3));
140
141     link3 = new Link<T>(links.GetLink(linkAddress3));
142
143     Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
144     Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
145
146     // Delete link
147     links.Delete(linkAddress3);
148
149     Assert.True(equalityComparer.Equals(links.Count(), two));
150
151     var setter3 = new Setter<T>(constants.Null);
152     links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
153
154     Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
155 }
156
157 public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
↪ links, int maximumOperationsPerCycle)
158 {
159     var comparer = Comparer<TLink>.Default;
160     var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
161     var uint64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
162     for (var N = 1; N < maximumOperationsPerCycle; N++)
163     {
164         var random = new System.Random(N);
165         var created = 0UL;
166         var deleted = 0UL;
167         for (var i = 0; i < N; i++)

```

```

168     {
169         var linksCount = addressToUInt64Converter.Convert(links.Count());
170         var createPoint = random.NextBoolean();
171         if (linksCount > 2 && createPoint)
172         {
173             var linksAddressRange = new Range<ulong>(1, linksCount);
174             TLink source = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA_
↳ ddressRange));
175             TLink target = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA_
↳ ddressRange));
176             ↪ //-V3086
177             var resultLink = links.GetOrCreate(source, target);
178             if (comparer.Compare(resultLink,
179                 ↪ uInt64ToAddressConverter.Convert(linksCount)) > 0)
180             {
181                 created++;
182             }
183         }
184         else
185         {
186             links.Create();
187             created++;
188         }
189     }
190     Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
191     for (var i = 0; i < N; i++)
192     {
193         TLink link = uInt64ToAddressConverter.Convert((ulong)i + 1UL);
194         if (links.Exists(link))
195         {
196             links.Delete(link);
197             deleted++;
198         }
199     }
200     Assert.True(addressToUInt64Converter.Convert(links.Count()) == 0L);
201 }
202 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.IO;
5  using System.Text;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Xunit;
9  using Platform.Disposables;
10 using Platform.Ranges;
11 using Platform.Random;
12 using Platform.Timestamps;
13 using Platform.Reflection;
14 using Platform.Singletons;
15 using Platform.Scopes;
16 using Platform.Counters;
17 using Platform.Diagnostics;
18 using Platform.IO;
19 using Platform.Memory;
20 using Platform.Data.Doublets.Decorators;
21 using Platform.Data.Doublets.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             {

```



```

38         new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeletions(100);
39     }
40 }
41
42 [Fact]
43 public static void CascadeUpdateTest()
44 {
45     var itself = _constants.Itself;
46     using (var scope = new TempLinksTestScope(useLog: true))
47     {
48         var links = scope.Links;
49
50         var l1 = links.Create();
51         var l2 = links.Create();
52
53         l2 = links.Update(l2, l2, l1, l2);
54
55         links.CreateAndUpdate(l2, itself);
56         links.CreateAndUpdate(l2, itself);
57
58         l2 = links.Update(l2, l1);
59
60         links.Delete(l2);
61
62         Global.Trash = links.Count();
63
64         links.Unsync.DisposeIfPossible(); // Close links to access log
65
66         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
67     }
68 }
69
70 [Fact]
71 public static void BasicTransactionLogTest()
72 {
73     using (var scope = new TempLinksTestScope(useLog: true))
74     {
75         var links = scope.Links;
76         var l1 = links.Create();
77         var l2 = links.Create();
78
79         Global.Trash = links.Update(l2, l2, l1, l2);
80
81         links.Delete(l1);
82
83         links.Unsync.DisposeIfPossible(); // Close links to access log
84
85         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
86     }
87 }
88
89 [Fact]
90 public static void TransactionAutoRevertedTest()
91 {
92     // Auto Reverted (Because no commit at transaction)
93     using (var scope = new TempLinksTestScope(useLog: true))
94     {
95         var links = scope.Links;
96         var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
97         using (var transaction = transactionsLayer.BeginTransaction())
98         {
99             var l1 = links.Create();
100             var l2 = links.Create();
101
102             links.Update(l2, l2, l1, l2);
103         }
104
105         Assert.Equal(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 [Fact]
114 public static void TransactionUserCodeErrorNoDataSavedTest()
115 {
116     // User Code Error (Autoreverted), no data saved
117     var itself = _constants.Itself;
118
119     TempLinksTestScope lastScope = null;
120     try
121     {
122         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
123             ↪ useLog: true))
124         {
125             var links = scope.Links;
126             var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor_
127             ↪ atorBase<ulong>)links.Unsync).Links;
128             using (var transaction = transactionsLayer.BeginTransaction())
129             {
130                 var l1 = links.CreateAndUpdate(itself, itself);
131                 var l2 = links.CreateAndUpdate(itself, itself);
132
133                 l2 = links.Update(l2, l2, l1, l2);
134
135                 links.CreateAndUpdate(l2, itself);
136                 links.CreateAndUpdate(l2, itself);
137
138                 //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi_
139                 ↪ tion>(scope.TempTransactionLogFilename);
140
141                 l2 = links.Update(l2, l1);
142
143                 links.Delete(l2);
144
145                 ExceptionThrower();
146
147                 transaction.Commit();
148             }
149             Global.Trash = links.Count();
150         }
151     } catch
152     {
153         Assert.False(lastScope == null);
154
155         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l_
156         ↪ astScope.TempTransactionLogFilename);
157
158         Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
159         ↪ transitions[0].After.IsNull());
160
161         lastScope.DeleteFiles();
162     }
163 }
164
165 [Fact]
166 public static void TransactionUserCodeErrorSomeDataSavedTest()
167 {
168     // User Code Error (Autoreverted), some data saved
169     var itself = _constants.Itself;
170
171     TempLinksTestScope lastScope = null;
172     try
173     {
174         using (var scope = new TempLinksTestScope(useLog: true))
175         {
176             var links = scope.Links;
177             l1 = links.CreateAndUpdate(itself, itself);
178             l2 = links.CreateAndUpdate(itself, itself);
179
180             l2 = links.Update(l2, l2, l1, l2);
181
182             links.CreateAndUpdate(l2, itself);
183             links.CreateAndUpdate(l2, itself);
184
185             links.Unsync.DisposeIfPossible();
186
187

```

```

188         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
189             ↪ scope.TempTransactionLogFilename);
190     }
191     using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
192         ↪ useLog: true))
193     {
194         var links = scope.Links;
195         var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
196         using (var transaction = transactionsLayer.BeginTransaction())
197         {
198             l2 = links.Update(l2, l1);
199             links.Delete(l2);
200             ExceptionThrower();
201             transaction.Commit();
202         }
203         Global.Trash = links.Count();
204     }
205 }
206 }
207 }
208 }
209 catch
210 {
211     Assert.False(lastScope == null);
212     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
213         ↪ Scope.TempTransactionLogFilename);
214     lastScope.DeleteFiles();
215 }
216 }
217 }
218 }
219 [Fact]
220 public static void TransactionCommit()
221 {
222     var itself = _constants.Itself;
223     var tempDatabaseFilename = Path.GetTempFileName();
224     var tempTransactionLogFilename = Path.GetTempFileName();
225     // Commit
226     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
227         ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
228         ↪ tempTransactionLogFilename))
229     using (var links = new UInt64Links(memoryAdapter))
230     {
231         using (var transaction = memoryAdapter.BeginTransaction())
232         {
233             var l1 = links.CreateAndUpdate(itself, itself);
234             var l2 = links.CreateAndUpdate(itself, itself);
235             Global.Trash = links.Update(l2, l2, l1, l2);
236             links.Delete(l1);
237             transaction.Commit();
238         }
239         Global.Trash = links.Count();
240     }
241     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
242         ↪ sactionLogFilename);
243 }
244 }
245 }
246 }
247 }
248 [Fact]
249 public static void TransactionDamage()
250 {
251     var itself = _constants.Itself;
252     var tempDatabaseFilename = Path.GetTempFileName();
253     var tempTransactionLogFilename = Path.GetTempFileName();
254     // Commit
255     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
256         ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
257         ↪ tempTransactionLogFilename))

```

```

259 using (var links = new UInt64Links(memoryAdapter))
260 {
261     using (var transaction = memoryAdapter.BeginTransaction())
262     {
263         var l1 = links.CreateAndUpdate(itself, itself);
264         var l2 = links.CreateAndUpdate(itself, itself);
265
266         Global.Trash = links.Update(l2, l2, l1, l2);
267
268         links.Delete(l1);
269
270         transaction.Commit();
271     }
272
273     Global.Trash = links.Count();
274 }
275
276 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTransactionLogFilename);
277
278 // Damage database
279
280 FileHelpers.WriteFirst(tempTransactionLogFilename, new
    ↳ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
281
282 // Try load damaged database
283 try
284 {
285     // TODO: Fix
286     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
        ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
        ↳ tempTransactionLogFilename))
287     using (var links = new UInt64Links(memoryAdapter))
288     {
289         Global.Trash = links.Count();
290     }
291 }
292 catch (NotSupportedException ex)
293 {
294     Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
        ↳ yet.");
295 }
296
297 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTransactionLogFilename);
298
299 File.Delete(tempDatabaseFilename);
300 File.Delete(tempTransactionLogFilename);
301 }
302
303 [Fact]
304 public static void Bug1Test()
305 {
306     var tempDatabaseFilename = Path.GetTempFileName();
307     var tempTransactionLogFilename = Path.GetTempFileName();
308
309     var itself = _constants.Itself;
310
311     // User Code Error (Autoreverted), some data saved
312     try
313     {
314         ulong l1;
315         ulong l2;
316
317         using (var memory = new UInt64ResizableDirectMemoryLinks(tempDatabaseFilename))
318         using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
            ↳ tempTransactionLogFilename))
319         using (var links = new UInt64Links(memoryAdapter))
320         {
321             l1 = links.CreateAndUpdate(itself, itself);
322             l2 = links.CreateAndUpdate(itself, itself);
323
324             l2 = links.Update(l2, l2, l1, l2);
325
326             links.CreateAndUpdate(l2, itself);
327             links.CreateAndUpdate(l2, itself);
328         }
329

```

```

330     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp ↵
    ↵ TransactionLogFilename);
331
332     using (var memory = new UInt64ResizableDirectMemoryLinks(tempDatabaseFilename))
333     using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
    ↵ tempTransactionLogFilename))
334     using (var links = new UInt64Links(memoryAdapter))
335     {
336         using (var transaction = memoryAdapter.BeginTransaction())
337         {
338             l2 = links.Update(l2, l1);
339
340             links.Delete(l2);
341
342             ExceptionThrower();
343
344             transaction.Commit();
345         }
346
347         Global.Trash = links.Count();
348     }
349 }
350 catch
351 {
352     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp ↵
    ↵ TransactionLogFilename);
353 }
354
355 File.Delete(tempDatabaseFilename);
356 File.Delete(tempTransactionLogFilename);
357 }
358
359 private static void ExceptionThrower() => throw new InvalidOperationException();
360
361 [Fact]
362 public static void PathsTest()
363 {
364     var source = _constants.SourcePart;
365     var target = _constants.TargetPart;
366
367     using (var scope = new TempLinksTestScope())
368     {
369         var links = scope.Links;
370         var l1 = links.CreatePoint();
371         var l2 = links.CreatePoint();
372
373         var r1 = links.GetByKeys(l1, source, target, source);
374         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
375     }
376 }
377
378 [Fact]
379 public static void RecursiveStringFormattingTest()
380 {
381     using (var scope = new TempLinksTestScope(useSequences: true))
382     {
383         var links = scope.Links;
384         var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
385
386         var a = links.CreatePoint();
387         var b = links.CreatePoint();
388         var c = links.CreatePoint();
389
390         var ab = links.GetOrCreate(a, b);
391         var cb = links.GetOrCreate(c, b);
392         var ac = links.GetOrCreate(a, c);
393
394         a = links.Update(a, c, b);
395         b = links.Update(b, a, c);
396         c = links.Update(c, a, b);
397
398         Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
399         Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
400         Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
401
402         Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
    ↵ "(5:(4:5 (6:5 4)) 6)");
403         Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
    ↵ "(6:(5:(4:5 6) 6) 4)");

```

```

404 Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
    ↳ "(4:(5:4 (6:5 4)) 6)");
405
406 // TODO: Think how to build balanced syntax tree while formatting structure (eg.
    ↳ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)"
407
408 Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
    ↳ "{5}{5}{4}{6}");
409 Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
    ↳ "{5}{6}{6}{4}");
410 Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
    ↳ "{4}{5}{4}{6}");
411 }
412 }
413
414 private static void DefaultFormatter(StringBuilder sb, ulong link)
415 {
416     sb.Append(link.ToString());
417 }
418
419 #endregion
420
421 #region Performance
422
423 /*
424 public static void RunAllPerformanceTests()
425 {
426     try
427     {
428         links.TestLinksInSteps();
429     }
430     catch (Exception ex)
431     {
432         ex.WriteToConsole();
433     }
434
435     return;
436
437     try
438     {
439         //ThreadPool.SetMaxThreads(2, 2);
440
441         // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
    ↳ результат
442         // Также это дополнительно помогает в отладке
443         // Увеличивает вероятность попадания информации в кэши
444         for (var i = 0; i < 10; i++)
445         {
446             //0 - 10 ГБ
447             //Каждые 100 МБ срез цифр
448
449             //links.TestGetSourceFunction();
450             //links.TestGetSourceFunctionInParallel();
451             //links.TestGetTargetFunction();
452             //links.TestGetTargetFunctionInParallel();
453             links.Create64BillionLinks();
454
455             links.TestRandomSearchFixed();
456             //links.Create64BillionLinksInParallel();
457             links.TestEachFunction();
458             //links.TestForeach();
459             //links.TestParallelForeach();
460         }
461
462         links.TestDeletionOfAllLinks();
463
464     }
465     catch (Exception ex)
466     {
467         ex.WriteToConsole();
468     }
469 }*/
470
471 /*
472 public static void TestLinksInSteps()
473 {
474     const long gibibyte = 1024 * 1024 * 1024;
475     const long mebibyte = 1024 * 1024;
476

```

```

477         var totalLinksToCreate = gibibyte /
↳ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
478         var linksStep = 102 * mebibyte /
↳ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
479
480         var creationMeasurements = new List<TimeSpan>();
481         var searchMeasurements = new List<TimeSpan>();
482         var deletionMeasurements = new List<TimeSpan>();
483
484         GetBaseRandomLoopOverhead(linksStep);
485         GetBaseRandomLoopOverhead(linksStep);
486
487         var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
488
489         ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
490
491         var loops = totalLinksToCreate / linksStep;
492
493         for (int i = 0; i < loops; i++)
494         {
495             creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
496             searchMeasurements.Add(Measure(() => links.RunRandomSearches(linksStep)));
497
498             Console.WriteLine("\rC + S {0}/{1}", i + 1, loops);
499         }
500
501         ConsoleHelpers.Debug();
502
503         for (int i = 0; i < loops; i++)
504         {
505             deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
506
507             Console.WriteLine("\rD {0}/{1}", i + 1, loops);
508         }
509
510         ConsoleHelpers.Debug();
511
512         ConsoleHelpers.Debug("C S D");
513
514         for (int i = 0; i < loops; i++)
515         {
516             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
↳ searchMeasurements[i], deletionMeasurements[i]);
517         }
518
519         ConsoleHelpers.Debug("C S D (no overhead)");
520
521         for (int i = 0; i < loops; i++)
522         {
523             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
↳ searchMeasurements[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
524         }
525
526         ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
↳ links.Total);
527     }
528
529     private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
↳ amountToCreate)
530     {
531         for (long i = 0; i < amountToCreate; i++)
532             links.Create(0, 0);
533     }
534
535     private static TimeSpan GetBaseRandomLoopOverhead(long loops)
536     {
537         return Measure(() =>
538         {
539             ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
540             ulong result = 0;
541             for (long i = 0; i < loops; i++)
542             {
543                 var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
544                 var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
545
546                 result += maxValue + source + target;
547             }
548             Global.Trash = result;
549         });

```

```

550     }
551     */
552
553     [Fact(Skip = "performance test")]
554     public static void GetSourceTest()
555     {
556         using (var scope = new TempLinksTestScope())
557         {
558             var links = scope.Links;
559             ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
560                 ↪ Iterations);
561
562             ulong counter = 0;
563
564             //var firstLink = links.First();
565             // Создаём одну связь, из которой будет производить считывание
566             var firstLink = links.Create();
567
568             var sw = Stopwatch.StartNew();
569
570             // Тестируем саму функцию
571             for (ulong i = 0; i < Iterations; i++)
572             {
573                 counter += links.GetSource(firstLink);
574             }
575
576             var elapsedTime = sw.Elapsed;
577
578             var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
579
580             // Удаляем связь, из которой производилось считывание
581             links.Delete(firstLink);
582
583             ConsoleHelpers.Debug(
584                 "{0} Iterations of GetSource function done in {1} ({2} Iterations per
585                 ↪ second), counter result: {3}",
586                 Iterations, elapsedTime, (long)iterationsPerSecond, counter);
587         }
588
589     [Fact(Skip = "performance test")]
590     public static void GetSourceInParallel()
591     {
592         using (var scope = new TempLinksTestScope())
593         {
594             var links = scope.Links;
595             ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
596                 ↪ parallel.", Iterations);
597
598             long counter = 0;
599
600             //var firstLink = links.First();
601             var firstLink = links.Create();
602
603             var sw = Stopwatch.StartNew();
604
605             // Тестируем саму функцию
606             Parallel.For(0, Iterations, x =>
607             {
608                 Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
609                 //Interlocked.Increment(ref counter);
610             });
611
612             var elapsedTime = sw.Elapsed;
613
614             var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
615
616             links.Delete(firstLink);
617
618             ConsoleHelpers.Debug(
619                 "{0} Iterations of GetSource function done in {1} ({2} Iterations per
620                 ↪ second), counter result: {3}",
621                 Iterations, elapsedTime, (long)iterationsPerSecond, counter);
622         }
623     }
624
625     [Fact(Skip = "performance test")]
626     public static void TestGetTarget()
627     {

```



```

625 using (var scope = new TempLinksTestScope())
626 {
627     var links = scope.Links;
628     ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
        ↪ Iterations);
629
630     ulong counter = 0;
631
632     //var firstLink = links.First();
633     var firstLink = links.Create();
634
635     var sw = Stopwatch.StartNew();
636
637     for (ulong i = 0; i < Iterations; i++)
638     {
639         counter += links.GetTarget(firstLink);
640     }
641
642     var elapsedTime = sw.Elapsed;
643
644     var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
645
646     links.Delete(firstLink);
647
648     ConsoleHelpers.Debug(
649         "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↪ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
650     }
651 }
652
653 [Fact(Skip = "performance test")]
654 public static void TestGetTargetInParallel()
655 {
656     using (var scope = new TempLinksTestScope())
657     {
658         var links = scope.Links;
659         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
        ↪ parallel.", Iterations);
660
661         long counter = 0;
662
663         //var firstLink = links.First();
664         var firstLink = links.Create();
665
666         var sw = Stopwatch.StartNew();
667
668         Parallel.For(0, Iterations, x =>
669         {
670             Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
671             //Interlocked.Increment(ref counter);
672         });
673
674         var elapsedTime = sw.Elapsed;
675
676         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
677
678         links.Delete(firstLink);
679
680         ConsoleHelpers.Debug(
681             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↪ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
682     }
683 }
684
685 // TODO: Заполнить базу данных перед тестом
686 /*
687 [Fact]
688 public void TestRandomSearchFixed()
689 {
690     var tempFilename = Path.GetTempFileName();
691
692     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
        ↪ DefaultLinksSizeStep))
693     {
694         long iterations = 64 * 1024 * 1024 /
        ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
695     }
696 }
697

```

```

698         ulong counter = 0;
699         var maxLink = links.Total;
700
701         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
702
703         var sw = Stopwatch.StartNew();
704
705         for (var i = iterations; i > 0; i--)
706         {
707             var source =
↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
708             var target =
↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
709
710             counter += links.Search(source, target);
711         }
712
713         var elapsedTime = sw.Elapsed;
714
715         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
716
717         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
↪ counter);
718     }
719
720     File.Delete(tempFilename);
721 }*/
722
723 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
724 public static void TestRandomSearchAll()
725 {
726     using (var scope = new TempLinksTestScope())
727     {
728         var links = scope.Links;
729         ulong counter = 0;
730
731         var maxLink = links.Count();
732
733         var iterations = links.Count();
734
735         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
↪ links.Count());
736
737         var sw = Stopwatch.StartNew();
738
739         for (var i = iterations; i > 0; i--)
740         {
741             var linksAddressRange = new
↪ Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);
742
743             var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
744             var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
745
746             counter += links.SearchOrDefault(source, target);
747         }
748
749         var elapsedTime = sw.Elapsed;
750
751         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
752
753         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪ Iterations per second), c: {3}",
↪ iterations, elapsedTime, (long)iterationsPerSecond, counter);
754     }
755 }
756
757 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
758 public static void TestEach()
759 {
760     using (var scope = new TempLinksTestScope())
761     {
762         var links = scope.Links;
763
764         var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
765
766         ConsoleHelpers.Debug("Testing Each function.");
767
768         var sw = Stopwatch.StartNew();
769
770

```

```

771         links.Each(counter.IncrementAndReturnTrue);
772
773         var elapsedTime = sw.Elapsed;
774
775         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
776
777         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
↪         ↪ links per second)",
778             counter, elapsedTime, (long)linksPerSecond);
779     }
780 }
781
782 /*
783 [Fact]
784 public static void TestForeach()
785 {
786     var tempFilename = Path.GetTempFileName();
787
788     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪     DefaultLinksSizeStep))
789     {
790         ulong counter = 0;
791
792         ConsoleHelpers.Debug("Testing foreach through links.");
793
794         var sw = Stopwatch.StartNew();
795
796         //foreach (var link in links)
797         //{
798             //    counter++;
799         //}
800
801         var elapsedTime = sw.Elapsed;
802
803         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
804
805         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
↪ links per second)", counter, elapsedTime, (long)linksPerSecond);
806     }
807
808     File.Delete(tempFilename);
809 }
810 */
811
812 /*
813 [Fact]
814 public static void TestParallelForeach()
815 {
816     var tempFilename = Path.GetTempFileName();
817
818     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪     DefaultLinksSizeStep))
819     {
820
821         long counter = 0;
822
823         ConsoleHelpers.Debug("Testing parallel foreach through links.");
824
825         var sw = Stopwatch.StartNew();
826
827         //Parallel.ForEach((IEnumerable<ulong>)links, x =>
828         //{
829             //    Interlocked.Increment(ref counter);
830         //});
831
832         var elapsedTime = sw.Elapsed;
833
834         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
835
836         ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↪ {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
837     }
838
839     File.Delete(tempFilename);
840 }
841 */
842
843 [Fact(Skip = "performance test")]
844 public static void Create64BillionLinks()
845 {

```

```

846     using (var scope = new TempLinksTestScope())
847     {
848         var links = scope.Links;
849         var linksBeforeTest = links.Count();
850
851         long linksToCreate = 64 * 1024 * 1024 /
            ↳ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
852
853         ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
854
855         var elapsedTime = Performance.Measure(() =>
856         {
857             for (long i = 0; i < linksToCreate; i++)
858             {
859                 links.Create();
860             }
861         });
862
863         var linksCreated = links.Count() - linksBeforeTest;
864         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
865
866         ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
867
868         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
            ↳ linksCreated, elapsedTime,
            ↳ (long)linksPerSecond);
869     }
870 }
871
872 [Fact(Skip = "performance test")]
873 public static void Create64BillionLinksInParallel()
874 {
875     using (var scope = new TempLinksTestScope())
876     {
877         var links = scope.Links;
878         var linksBeforeTest = links.Count();
879
880         var sw = Stopwatch.StartNew();
881
882         long linksToCreate = 64 * 1024 * 1024 /
883             ↳ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
884
885         ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
886
887         Parallel.For(0, linksToCreate, x => links.Create());
888
889         var elapsedTime = sw.Elapsed;
890
891         var linksCreated = links.Count() - linksBeforeTest;
892         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
893
894         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
            ↳ linksCreated, elapsedTime,
            ↳ (long)linksPerSecond);
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)",
            ↳ linksDeleted, elapsedTime,
            ↳ (long)linksPerSecond);
914     }
915 }
916
917 }
918
919 #endregion
920

```

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

```

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

```

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

```

1  using Xunit;
2  using Platform.Converters;
3  using Platform.Memory;
4  using Platform.Reflection;
5  using Platform.Scopes;
6  using Platform.Data.Numbers.Raw;
7  using Platform.Data.Doublets.Incrementers;
8  using Platform.Data.Doublets.Numbers.Unary;
9  using Platform.Data.Doublets.PropertyOperators;
10 using Platform.Data.Doublets.Sequences.Converters;
11 using Platform.Data.Doublets.Sequences.Indexes;
12 using Platform.Data.Doublets.Sequences.Walkers;
13 using Platform.Data.Doublets.Unicode;
14 using Platform.Data.Doublets.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);

```

```

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

```

```

85     var linkToItsFrequencyNumberConverter = new
      ↳ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
      ↳ unaryNumberToAddressConverter);
86     var sequenceToItsLocalElementLevelsConverter = new
      ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
      ↳ linkToItsFrequencyNumberConverter);
87     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
      ↳ sequenceToItsLocalElementLevelsConverter);
88
89     var stringToUnicodeSequenceConverter = new
      ↳ StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
      ↳ index, optimalVariantConverter, unicodeSequenceMarker);
90
91     var originalString = "Hello";
92
93     var unicodeSequenceLink =
      ↳ stringToUnicodeSequenceConverter.Convert(originalString);
94
95     var unicodeSymbolCriterionMatcher = new
      ↳ UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
96     var unicodeSymbolToCharConverter = new
      ↳ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
      ↳ unicodeSymbolCriterionMatcher);
97
98     var unicodeSequenceCriterionMatcher = new
      ↳ UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
99
100    var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
      ↳ unicodeSymbolCriterionMatcher.IsMatched);
101
102    var unicodeSequenceToStringConverter = new
      ↳ UnicodeSequenceToStringConverter<ulong>(links,
      ↳ unicodeSequenceCriterionMatcher, sequenceWalker,
      ↳ unicodeSymbolToCharConverter);
103
104    var resultingString =
      ↳ unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
105
106    Assert.Equal(originalString, resultingString);
107    }
108  }
109 }
110

```

Index

`./csharp/Platform.Data.Doublets.Tests/ComparisonTests.cs`, 164
`./csharp/Platform.Data.Doublets.Tests/EqualityTests.cs`, 165
`./csharp/Platform.Data.Doublets.Tests/GenericLinksTests.cs`, 167
`./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs`, 167
`./csharp/Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs`, 168
`./csharp/Platform.Data.Doublets.Tests/ReadSequenceTests.cs`, 171
`./csharp/Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs`, 172
`./csharp/Platform.Data.Doublets.Tests/ScopeTests.cs`, 173
`./csharp/Platform.Data.Doublets.Tests/SequencesTests.cs`, 173
`./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs`, 188
`./csharp/Platform.Data.Doublets.Tests/TempLinksTestScope.cs`, 189
`./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs`, 189
`./csharp/Platform.Data.Doublets.Tests/UInt64LinksTests.cs`, 192
`./csharp/Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs`, 205
`./csharp/Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs`, 205
`./csharp/Platform.Data.Doublets/Decorators/LinksCascadeUniquenessAndUsagesResolver.cs`, 1
`./csharp/Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.cs`, 1
`./csharp/Platform.Data.Doublets/Decorators/LinksDecoratorBase.cs`, 1
`./csharp/Platform.Data.Doublets/Decorators/LinksDisposableDecoratorBase.cs`, 2
`./csharp/Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.cs`, 3
`./csharp/Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.cs`, 3
`./csharp/Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.cs`, 4
`./csharp/Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.cs`, 4
`./csharp/Platform.Data.Doublets/Decorators/LinksUniquenessResolver.cs`, 4
`./csharp/Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs`, 5
`./csharp/Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs`, 5
`./csharp/Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs`, 6
`./csharp/Platform.Data.Doublets/Decorators/UInt64Links.cs`, 6
`./csharp/Platform.Data.Doublets/Decorators/UniLinks.cs`, 7
`./csharp/Platform.Data.Doublets/Doublet.cs`, 12
`./csharp/Platform.Data.Doublets/DoubletComparer.cs`, 12
`./csharp/Platform.Data.Doublets/ILinks.cs`, 13
`./csharp/Platform.Data.Doublets/ILinksExtensions.cs`, 13
`./csharp/Platform.Data.Doublets/ISynchronizedLinks.cs`, 24
`./csharp/Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs`, 25
`./csharp/Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs`, 25
`./csharp/Platform.Data.Doublets/Link.cs`, 26
`./csharp/Platform.Data.Doublets/LinkExtensions.cs`, 29
`./csharp/Platform.Data.Doublets/LinksOperatorBase.cs`, 29
`./csharp/Platform.Data.Doublets/Memory/ILinksListMethods.cs`, 29
`./csharp/Platform.Data.Doublets/Memory/ILinksTreeMethods.cs`, 30
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/LinksSizeBalancedTreeMethodsBase.cs`, 30
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/LinksSourcesSizeBalancedTreeMethods.cs`, 32
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/LinksTargetsSizeBalancedTreeMethods.cs`, 33
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinks.cs`, 34
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinksBase.cs`, 35
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/UnusedLinksListMethods.cs`, 43
`./csharp/Platform.Data.Doublets/Memory/Split/LinksHeaderIndexPart.cs`, 44
`./csharp/Platform.Data.Doublets/Memory/Split/RawLinkDataPart.cs`, 44
`./csharp/Platform.Data.Doublets/Memory/Split/RawLinkIndexPart.cs`, 45
`./csharp/Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs`, 46
`./csharp/Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConverter.cs`, 46
`./csharp/Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs`, 47
`./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs`, 47
`./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs`, 49
`./csharp/Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs`, 49
`./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs`, 50
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksAvlBalancedTreeMethodsBase.cs`, 51
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSizeBalancedTreeMethodsBase.cs`, 55
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesAvlBalancedTreeMethods.cs`, 58
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesSizeBalancedTreeMethods.cs`, 60
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsAvlBalancedTreeMethods.cs`, 61
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsSizeBalancedTreeMethods.cs`, 62
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinks.cs`, 63
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinksBase.cs`, 64
`./csharp/Platform.Data.Doublets/ResizableDirectMemory/Generic/UnusedLinksListMethods.cs`, 71

./csharp/Platform.Data.Doublets/ResizableDirectMemory/ILinksListMethods.cs, 72
./csharp/Platform.Data.Doublets/ResizableDirectMemory/ILinksTreeMethods.cs, 72
./csharp/Platform.Data.Doublets/ResizableDirectMemory/LinksHeader.cs, 72
./csharp/Platform.Data.Doublets/ResizableDirectMemory/RawLink.cs, 73
./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs, 74
./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs, 75
./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs, 77
./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.cs, 78
./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs, 79
./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs, 80
./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64ResizableDirectMemoryLinks.cs, 81
./csharp/Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64UnusedLinksListMethods.cs, 83
./csharp/Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 83
./csharp/Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 84
./csharp/Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 87
./csharp/Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 87
./csharp/Platform.Data.Doublets/Sequences/Converters/SequenceToToltsLocalElementLevelsConverter.cs, 89
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs, 89
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs, 90
./csharp/Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 90
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 91
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 91
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 94
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 96
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToltsFrequencyValueConverter.cs, 96
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 96
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 97
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 97
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 98
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 98
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 98
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 99
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 100
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 101
./csharp/Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 101
./csharp/Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 102
./csharp/Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 102
./csharp/Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 103
./csharp/Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 103
./csharp/Platform.Data.Doublets/Sequences/Indexes/Unindex.cs, 104
./csharp/Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 104
./csharp/Platform.Data.Doublets/Sequences/Sequences.cs, 131
./csharp/Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 142
./csharp/Platform.Data.Doublets/Sequences/SequencesOptions.cs, 143
./csharp/Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 145
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 146
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 146
./csharp/Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 148
./csharp/Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 148
./csharp/Platform.Data.Doublets/Stacks/Stack.cs, 149
./csharp/Platform.Data.Doublets/Stacks/StackExtensions.cs, 150
./csharp/Platform.Data.Doublets/SynchronizedLinks.cs, 150
./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs, 151
./csharp/Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 153
./csharp/Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 159
./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 159
./csharp/Platform.Data.Doublets/Unicode/UnicodeMap.cs, 160
./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs, 163
./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 163
./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs, 164
./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 164