

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;
52         }
53     }
54 }

```

```

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     }
255 }

```

```

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,
        ↪ links.Constants.Continue);
658         links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
659         for (var i = count - 1; i >= 0; i--)
660         {
661             links.Delete(queryResult[i]);
662         }
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(ICollection<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/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.Memory
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 RootAsSource;
22         public TLink RootAsTarget;
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(RootAsSource, other.RootAsSource)
37             && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
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, RootAsSource, RootAsTarget, LastFreeLink, Reserved8).GetHashCode();
44     }
45 }

```

```

42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     public static bool operator ==(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
44         ↪ left.Equals(right);
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     public static bool operator !=(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
48         ↪ !(left == right);
49 }
50 }

```

1.28 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSizeBalancedTreeMethodsBase.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 ExternalLinksSizeBalancedTreeMethodsBase<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 ExternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants,
27             ↪ byte* 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();
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
44             ↪ rootSource, TLink rootTarget);
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
48             ↪ rootSource, TLink rootTarget);
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
52             ↪ AsRef<LinksHeader<TLink>>(Header);
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
56             ↪ AsRef<RawLinkDataPart<TLink>>(LinksDataParts + RawLinkDataPart<TLink>.SizeInBytes *
57             ↪ _addressToInt64Converter.Convert(link));
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
61             ↪ ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
62             ↪ RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link));
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
66         {
67             ref var link = ref GetLinkDataPartReference(linkIndex);
68             return new Link<TLink>(linkIndex, link.Source, link.Target);
69         }
70     }
71 }

```

```

60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
62 {
63     ref var firstLink = ref GetLinkDataPartReference(first);
64     ref var secondLink = ref GetLinkDataPartReference(second);
65     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
66         ↪ 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 GetLinkDataPartReference(first);
73     ref var secondLink = ref GetLinkDataPartReference(second);
74     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
75         ↪ secondLink.Source, secondLink.Target);
76 }
77
78 public TLink this[TLink index]
79 {
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     get
82     {
83         var root = GetTreeRoot();
84         if (GreaterOrEqualThan(index, GetSize(root)))
85         {
86             return Zero;
87         }
88         while (!EqualToZero(root))
89         {
90             var left = GetLeftOrDefault(root);
91             var leftSize = GetSizeOrZero(left);
92             if (LessThan(index, leftSize))
93             {
94                 root = left;
95                 continue;
96             }
97             if (AreEqual(index, leftSize))
98             {
99                 return root;
100             }
101             root = GetRightOrDefault(root);
102             index = Subtract(index, Increment(leftSize));
103         }
104         return Zero; // TODO: Impossible situation exception (only if tree structure
105             ↪ broken)
106     }
107 }
108
109 /// <summary>
110 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
111 /// (концом).
112 /// </summary>
113 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
114 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
115 /// <returns>Индекс искомой связи.</returns>
116 [MethodImpl(MethodImplOptions.AggressiveInlining)]
117 public TLink Search(TLink source, TLink target)
118 {
119     var root = GetTreeRoot();
120     while (!EqualToZero(root))
121     {
122         ref var rootLink = ref GetLinkDataPartReference(root);
123         var rootSource = rootLink.Source;
124         var rootTarget = rootLink.Target;
125         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
126             ↪ node.Key < root.Key
127         {
128             root = GetLeftOrDefault(root);
129         }
130         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
131             ↪ node.Key > root.Key
132         {
133             root = GetRightOrDefault(root);
134         }
135         else // node.Key == root.Key
136         {
137             return root;
138         }
139     }
140 }

```



```

132         return root;
133     }
134 }
135 return Zero;
136 }
137
138 // TODO: Return indices range instead of references count
139 [MethodImpl(MethodImplOptions.AggressiveInlining)]
140 public TLink CountUsages(TLink link)
141 {
142     var root = GetTreeRoot();
143     var total = GetSize(root);
144     var totalRightIgnore = Zero;
145     while (!EqualToZero(root))
146     {
147         var @base = GetBasePartValue(root);
148         if (LessOrEqualThan(@base, link))
149         {
150             root = GetRightOrDefault(root);
151         }
152         else
153         {
154             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
155             root = GetLeftOrDefault(root);
156         }
157     }
158     root = GetTreeRoot();
159     var totalLeftIgnore = Zero;
160     while (!EqualToZero(root))
161     {
162         var @base = GetBasePartValue(root);
163         if (GreaterOrEqualThan(@base, link))
164         {
165             root = GetLeftOrDefault(root);
166         }
167         else
168         {
169             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
170             root = GetRightOrDefault(root);
171         }
172     }
173     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
174 }
175
176 [MethodImpl(MethodImplOptions.AggressiveInlining)]
177 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
178     ↳ EachUsageCore(@base, GetTreeRoot(), handler);
179
180 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
181 ↳ low-level MSIL stack.
182 [MethodImpl(MethodImplOptions.AggressiveInlining)]
183 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
184 {
185     var @continue = Continue;
186     if (EqualToZero(link))
187     {
188         return @continue;
189     }
190     var linkBasePart = GetBasePartValue(link);
191     var @break = Break;
192     if (GreaterThan(linkBasePart, @base))
193     {
194         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
195         {
196             return @break;
197         }
198     }
199     else if (LessThan(linkBasePart, @base))
200     {
201         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
202         {
203             return @break;
204         }
205     }
206     else //if (linkBasePart == @base)
207     {
208         if (AreEqual(handler(GetLinkValues(link)), @break))
209         {
210             return @break;
211         }
212     }
213 }

```

```

209     }
210     if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
211     {
212         return @break;
213     }
214     if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
215     {
216         return @break;
217     }
218 }
219 return @continue;
220 }
221
222 [MethodImpl(MethodImplOptions.AggressiveInlining)]
223 protected override void PrintNodeValue(TLink node, StringBuilder sb)
224 {
225     ref var link = ref GetLinkDataPartReference(node);
226     sb.Append(' ');
227     sb.Append(link.Source);
228     sb.Append('-');
229     sb.Append('>');
230     sb.Append(link.Target);
231 }
232 }
233 }

```

1.29 ./csharp/Platform.Data.Doublets.Memory.Split.Generic.ExternalLinksSourcesSizeBalancedTreeMethods.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 ExternalLinksSourcesSizeBalancedTreeMethods<TLink> :
8         ↳ ExternalLinksSizeBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public ExternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants,
12             ↳ byte* 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).LeftAsSource;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected override ref TLink GetRightReference(TLink node) => ref
21             ↳ GetLinkIndexPartReference(node).RightAsSource;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override TLink GetLeft(TLink node) =>
25             ↳ GetLinkIndexPartReference(node).LeftAsSource;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override TLink GetRight(TLink node) =>
29             ↳ GetLinkIndexPartReference(node).RightAsSource;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetLeft(TLink node, TLink left) =>
33             ↳ GetLinkIndexPartReference(node).LeftAsSource = left;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override void SetRight(TLink node, TLink right) =>
37             ↳ GetLinkIndexPartReference(node).RightAsSource = right;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override TLink GetSize(TLink node) =>
41             ↳ GetLinkIndexPartReference(node).SizeAsSource;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override void SetSize(TLink node, TLink size) =>
45             ↳ GetLinkIndexPartReference(node).SizeAsSource = size;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override TLink GetBasePartValue(TLink link) =>
52             ↳ GetLinkDataPartReference(link).Source;
53     }
54 }

```

```

41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
43     ↪ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
44     ↪ AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget);
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
48     ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
49     ↪ AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget);
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override void ClearNode(TLink node)
53 {
54     ref var link = ref GetLinkIndexPartReference(node);
55     link.LeftAsSource = Zero;
56     link.RightAsSource = Zero;
57     link.SizeAsSource = Zero;
58 }
59 }

```

1.30 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsSizeBalancedTreeMethods.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 ExternalLinksTargetsSizeBalancedTreeMethods<TLink> :
8         ↪ ExternalLinksSizeBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public ExternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants,
12             ↪ byte* 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 TLink GetSize(TLink node) =>
41             ↪ GetLinkIndexPartReference(node).SizeAsTarget;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override void SetSize(TLink node, TLink size) =>
45             ↪ GetLinkIndexPartReference(node).SizeAsTarget = size;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override TLink GetBasePartValue(TLink link) =>
52             ↪ GetLinkDataPartReference(link).Target;
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

43     protected override bool FirstIsToLeftOfSecond(TLink firstSource, TLink firstTarget,
44         ↪ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
45         ↪ AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource);
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
49         ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
50         ↪ AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource);
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected override void ClearNode(TLink node)
54     {
55         ref var link = ref GetLinkIndexPartReference(node);
56         link.LeftAsTarget = Zero;
57         link.RightAsTarget = Zero;
58         link.SizeAsTarget = Zero;
59     }
60 }

```

1.31 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSizeBalancedTreeMethodsBase.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 InternalLinksSizeBalancedTreeMethodsBase<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 InternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants,
27             ↪ byte* 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 abstract TLink GetKeyPartValue(TLink link);
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
47             ↪ AsRef<RawLinkDataPart<TLink>>(LinksDataParts + RawLinkDataPart<TLink>.SizeInBytes *
48             ↪ _addressToInt64Converter.Convert(link));
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
52             ↪ ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
53             ↪ RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link));
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override bool FirstIsToLeftOfSecond(TLink first, TLink second) =>
57             ↪ LessThan(GetKeyPartValue(first), GetKeyPartValue(second));
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

52     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
53         ↪ GreaterThan(GetKeyPartValue(first), GetKeyPartValue(second));
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
57     {
58         ref var link = ref GetLinkDataPartReference(linkIndex);
59         return new Link<TLink>(linkIndex, link.Source, link.Target);
60     }
61
62     public TLink this[TLink link, TLink index]
63     {
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         get
66         {
67             var root = GetTreeRoot(link);
68             if (GreaterOrEqualThan(index, GetSize(root)))
69             {
70                 return Zero;
71             }
72             while (!EqualToZero(root))
73             {
74                 var left = GetLeftOrDefault(root);
75                 var leftSize = GetSizeOrZero(left);
76                 if (LessThan(index, leftSize))
77                 {
78                     root = left;
79                     continue;
80                 }
81                 if (AreEqual(index, leftSize))
82                 {
83                     return root;
84                 }
85                 root = GetRightOrDefault(root);
86                 index = Subtract(index, Increment(leftSize));
87             }
88             return Zero; // TODO: Impossible situation exception (only if tree structure
89                 ↪ broken)
90         }
91     }
92
93     /// <summary>
94     /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
95     /// ↪ (концом).
96     /// </summary>
97     /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
98     /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
99     /// <returns>Индекс искомой связи.</returns>
100     [MethodImpl(MethodImplOptions.AggressiveInlining)]
101     public abstract TLink Search(TLink source, TLink target);
102
103     [MethodImpl(MethodImplOptions.AggressiveInlining)]
104     protected TLink SearchCore(TLink root, TLink key)
105     {
106         while (!EqualToZero(root))
107         {
108             var rootKey = GetKeyPartValue(root);
109             if (LessThan(key, rootKey)) // node.Key < root.Key
110             {
111                 root = GetLeftOrDefault(root);
112             }
113             else if (GreaterThan(key, rootKey)) // node.Key > root.Key
114             {
115                 root = GetRightOrDefault(root);
116             }
117             else // node.Key == root.Key
118             {
119                 return root;
120             }
121         }
122         return Zero;
123     }
124
125     // TODO: Return indices range instead of references count
126     [MethodImpl(MethodImplOptions.AggressiveInlining)]
127     public TLink CountUsages(TLink link) => GetSizeOrZero(GetTreeRoot(link));
128
129     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

1.32 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesSizeBalancedTreeMethods.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 InternalLinksSourcesSizeBalancedTreeMethods<TLink> :
8         ↳ InternalLinksSizeBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants,
12             ↳ byte* 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).LeftAsSource;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected override ref TLink GetRightReference(TLink node) => ref
21             ↳ GetLinkIndexPartReference(node).RightAsSource;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override TLink GetLeft(TLink node) =>
25             ↳ GetLinkIndexPartReference(node).LeftAsSource;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override TLink GetRight(TLink node) =>
29             ↳ GetLinkIndexPartReference(node).RightAsSource;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetLeft(TLink node, TLink left) =>
33             ↳ GetLinkIndexPartReference(node).LeftAsSource = left;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

1.33 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsSizeBalancedTreeMethods.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 InternalLinksTargetsSizeBalancedTreeMethods<TLink> :
8          ↪ InternalLinksSizeBalancedTreeMethodsBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public InternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants,
12             ↪ byte* 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)]

```

```

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

```

1.34 ./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>> _createInternalSourceTreeMethods;
14         private readonly Func<ILinksTreeMethods<TLink>> _createExternalSourceTreeMethods;
15         private readonly Func<ILinksTreeMethods<TLink>> _createInternalTargetTreeMethods;
16         private readonly Func<ILinksTreeMethods<TLink>> _createExternalTargetTreeMethods;
17         private byte* _header;
18         private byte* _linksDataParts;
19         private byte* _linksIndexParts;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
23             ↪ indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
27             ↪ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
28             ↪ memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
32             ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
33             ↪ base(dataMemory, indexMemory, memoryReservationStep, constants)
34         {
35             _createInternalSourceTreeMethods = () => new
36                 ↪ InternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
37                 ↪ _linksIndexParts, _header);
38             _createExternalSourceTreeMethods = () => new
39                 ↪ ExternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
40                 ↪ _linksIndexParts, _header);
41             _createInternalTargetTreeMethods = () => new
42                 ↪ InternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
43                 ↪ _linksIndexParts, _header);
44         }
45     }
46 }

```



```

33     _createExternalTargetTreeMethods = () => new
34         ↪ ExternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
35         ↪ _linksIndexParts, _header);
36     Init(dataMemory, indexMemory, memoryReservationStep);
37 }
38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 protected override void SetPointers(IResizableDirectMemory dataMemory,
40     ↪ IResizableDirectMemory indexMemory)
41 {
42     _linksDataParts = (byte*)dataMemory.Pointer;
43     _linksIndexParts = (byte*)indexMemory.Pointer;
44     _header = _linksIndexParts;
45     InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
46     ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
47     InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
48     ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
49     UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_linksDataParts, _header);
50 }
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override void ResetPointers()
53 {
54     base.ResetPointers();
55     _linksDataParts = null;
56     _linksIndexParts = null;
57     _header = null;
58 }
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected override ref LinksHeader<TLink> GetHeaderReference() => ref
61     ↪ AsRef<LinksHeader<TLink>>(_header);
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
64     ↪ => ref AsRef<RawLinkDataPart<TLink>>(_linksDataParts + LinkDataPartSizeInBytes *
65     ↪ ConvertToInt64(linkIndex));
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
68     ↪ linkIndex) => ref AsRef<RawLinkIndexPart<TLink>>(_linksIndexParts +
69     ↪ LinkIndexPartSizeInBytes * ConvertToInt64(linkIndex));
70 }
71 }

```

1.35 ./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 =
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 LinkDataPartSizeInBytes = RawLinkDataPart<TLink>.SizeInBytes;

```

```

32 public static readonly long LinkIndexPartSizeInBytes =
    ↳ RawLinkIndexPart<TLink>.SizeInBytes;
33
34 public static readonly long LinkHeaderSizeInBytes = LinksHeader<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> InternalSourcesTreeMethods;
44 protected ILinksTreeMethods<TLink> ExternalSourcesTreeMethods;
45 protected ILinksTreeMethods<TLink> InternalTargetsTreeMethods;
46 protected ILinksTreeMethods<TLink> ExternalTargetsTreeMethods;
47 // 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
48 protected ILinksListMethods<TLink> UnusedLinksListMethods;
49
50 /// <summary>
51 /// Возвращает общее число связей находящихся в хранилище.
52 /// </summary>
53 protected virtual TLink Total
54 {
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     get
57     {
58         ref var header = ref GetHeaderReference();
59         return Subtract(header.AllocatedLinks, header.FreeLinks);
60     }
61 }
62
63 public virtual LinksConstants<TLink> Constants
64 {
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     get;
67 }
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants)
71 {
72     _dataMemory = dataMemory;
73     _indexMemory = indexMemory;
74     _dataMemoryReservationStepInBytes = memoryReservationStep * LinkDataPartSizeInBytes;
75     _indexMemoryReservationStepInBytes = memoryReservationStep *
    ↳ LinkIndexPartSizeInBytes;
76     Constants = constants;
77 }
78
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
    ↳ memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }
81
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 protected virtual void Init(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory, long memoryReservationStep)
84 {
85     if (dataMemory.ReservedCapacity < memoryReservationStep)
86     {
87         dataMemory.ReservedCapacity = memoryReservationStep;
88     }
89     if (indexMemory.ReservedCapacity < memoryReservationStep)
90     {
91         indexMemory.ReservedCapacity = memoryReservationStep;
92     }
93     SetPointers(dataMemory, indexMemory);
94     ref var header = ref GetHeaderReference();
95     // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
96     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
97     dataMemory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) *
    ↳ LinkDataPartSizeInBytes + LinkDataPartSizeInBytes; // First link is read only
    ↳ zero link.
98     indexMemory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) *
    ↳ LinkIndexPartSizeInBytes + LinkHeaderSizeInBytes;
99     // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
100    // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity

```

```

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

```

```

173         if (externalReferencesRange.HasValue &&
174             ⇨ externalReferencesRange.Value.Contains(target))
175         {
176             return ExternalTargetsTreeMethods.CountUsages(target);
177         }
178         else
179         {
180             return InternalTargetsTreeMethods.CountUsages(target);
181         }
182     }
183     else if (AreEqual(target, any))
184     {
185         if (externalReferencesRange.HasValue &&
186             ⇨ externalReferencesRange.Value.Contains(source))
187         {
188             return ExternalSourcesTreeMethods.CountUsages(source);
189         }
190         else
191         {
192             return InternalSourcesTreeMethods.CountUsages(source);
193         }
194     }
195     // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
196     TLink link;
197     if (externalReferencesRange.HasValue)
198     {
199         if (externalReferencesRange.Value.Contains(source) &&
200             ⇨ externalReferencesRange.Value.Contains(target))
201         {
202             link = ExternalSourcesTreeMethods.Search(source, target);
203         }
204         else if (externalReferencesRange.Value.Contains(source))
205         {
206             link = InternalTargetsTreeMethods.Search(source, target);
207         }
208         else if (externalReferencesRange.Value.Contains(target))
209         {
210             link = InternalSourcesTreeMethods.Search(source, target);
211         }
212         else
213         {
214             if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
215                 ⇨ InternalTargetsTreeMethods.CountUsages(target)))
216             {
217                 link = InternalTargetsTreeMethods.Search(source, target);
218             }
219             else
220             {
221                 link = InternalSourcesTreeMethods.Search(source, target);
222             }
223         }
224     }
225     else
226     {
227         if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
228             ⇨ InternalTargetsTreeMethods.CountUsages(target)))
229         {
230             link = InternalTargetsTreeMethods.Search(source, target);
231         }
232         else
233         {
234             link = InternalSourcesTreeMethods.Search(source, target);
235         }
236     }
237     return AreEqual(link, constants.Null) ? GetZero() : GetOne();
238 }
239 else
240 {
241     if (!Exists(index))
242     {
243         return GetZero();
244     }
245     if (AreEqual(source, any) && AreEqual(target, any))
246     {
247         return GetOne();
248     }

```

```

246     }
247     ref var storedLinkValue = ref GetLinkDataPartReference(index);
248     if (!AreEqual(source, any) && !AreEqual(target, any))
249     {
250         if (AreEqual(storedLinkValue.Source, source) &&
251             ↪ AreEqual(storedLinkValue.Target, target))
252         {
253             return GetOne();
254         }
255         return GetZero();
256     }
257     var value = default(TLink);
258     if (AreEqual(source, any))
259     {
260         value = target;
261     }
262     if (AreEqual(target, any))
263     {
264         value = source;
265     }
266     if (AreEqual(storedLinkValue.Source, value) ||
267         ↪ AreEqual(storedLinkValue.Target, value))
268     {
269         return GetOne();
270     }
271     return GetZero();
272 }
273 }
274
275 [MethodImpl(MethodImplOptions.AggressiveInlining)]
276 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
277 {
278     var constants = Constants;
279     var @break = constants.Break;
280     if (restrictions.Count == 0)
281     {
282         for (var link = GetOne(); LessOrEqualThan(link,
283             ↪ GetHeaderReference().AllocatedLinks); link = Increment(link))
284         {
285             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
286             {
287                 return @break;
288             }
289         }
290         return @break;
291     }
292     var @continue = constants.Continue;
293     var any = constants.Any;
294     var index = restrictions[constants.IndexPart];
295     if (restrictions.Count == 1)
296     {
297         if (AreEqual(index, any))
298         {
299             return Each(handler, Array.Empty<TLink>());
300         }
301         if (!Exists(index))
302         {
303             return @continue;
304         }
305         return handler(GetLinkStruct(index));
306     }
307     if (restrictions.Count == 2)
308     {
309         var value = restrictions[1];
310         if (AreEqual(index, any))
311         {
312             if (AreEqual(value, any))
313             {
314                 return Each(handler, Array.Empty<TLink>());
315             }
316             if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
317             {
318                 return @break;
319             }
320             return Each(handler, new Link<TLink>(index, any, value));
321         }
322     }

```

```

320     }
321     else
322     {
323         if (!Exists(index))
324         {
325             return @continue;
326         }
327         if (AreEqual(value, any))
328         {
329             return handler(GetLinkStruct(index));
330         }
331         ref var storedLinkValue = ref GetLinkDataPartReference(index);
332         if (AreEqual(storedLinkValue.Source, value) ||
333             AreEqual(storedLinkValue.Target, value))
334         {
335             return handler(GetLinkStruct(index));
336         }
337         return @continue;
338     }
339 }
340 if (restrictions.Count == 3)
341 {
342     var externalReferencesRange = constants.ExternalReferencesRange;
343     var source = restrictions[constants.SourcePart];
344     var target = restrictions[constants.TargetPart];
345     if (AreEqual(index, any))
346     {
347         if (AreEqual(source, any) && AreEqual(target, any))
348         {
349             return Each(handler, Array.Empty<TLink>());
350         }
351         else if (AreEqual(source, any))
352         {
353             if (externalReferencesRange.HasValue &&
354                 ↪ externalReferencesRange.Value.Contains(target))
355             {
356                 return ExternalTargetsTreeMethods.EachUsage(target, handler);
357             }
358             else
359             {
360                 return InternalTargetsTreeMethods.EachUsage(target, handler);
361             }
362         }
363         else if (AreEqual(target, any))
364         {
365             if (externalReferencesRange.HasValue &&
366                 ↪ externalReferencesRange.Value.Contains(source))
367             {
368                 return ExternalSourcesTreeMethods.EachUsage(source, handler);
369             }
370             else
371             {
372                 return InternalSourcesTreeMethods.EachUsage(source, handler);
373             }
374         }
375         else //if(source != Any && target != Any)
376         {
377             TLink link;
378             if (externalReferencesRange.HasValue)
379             {
380                 if (externalReferencesRange.Value.Contains(source) &&
381                     ↪ externalReferencesRange.Value.Contains(target))
382                 {
383                     link = ExternalSourcesTreeMethods.Search(source, target);
384                 }
385                 else if (externalReferencesRange.Value.Contains(source))
386                 {
387                     link = InternalTargetsTreeMethods.Search(source, target);
388                 }
389                 else if (externalReferencesRange.Value.Contains(target))
390                 {
391                     link = InternalSourcesTreeMethods.Search(source, target);
392                 }
393                 else
394                 {
395                     if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
396                         ↪ InternalTargetsTreeMethods.CountUsages(target)))
397                     {

```

```

394         link = InternalTargetsTreeMethods.Search(source, target);
395     }
396     else
397     {
398         link = InternalSourcesTreeMethods.Search(source, target);
399     }
400 }
401 }
402 else
403 {
404     if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
405         ↪ InternalTargetsTreeMethods.CountUsages(target)))
406     {
407         link = InternalTargetsTreeMethods.Search(source, target);
408     }
409     else
410     {
411         link = InternalSourcesTreeMethods.Search(source, target);
412     }
413     return AreEqual(link, constants.Null) ? @continue :
414         ↪ handler(GetLinkStruct(link));
415 }
416 else
417 {
418     if (!Exists(index))
419     {
420         return @continue;
421     }
422     if (AreEqual(source, any) && AreEqual(target, any))
423     {
424         return handler(GetLinkStruct(index));
425     }
426     ref var storedLinkValue = ref GetLinkDataPartReference(index);
427     if (!AreEqual(source, any) && !AreEqual(target, any))
428     {
429         if (AreEqual(storedLinkValue.Source, source) &&
430             AreEqual(storedLinkValue.Target, target))
431         {
432             return handler(GetLinkStruct(index));
433         }
434         return @continue;
435     }
436     var value = default(TLink);
437     if (AreEqual(source, any))
438     {
439         value = target;
440     }
441     if (AreEqual(target, any))
442     {
443         value = source;
444     }
445     if (AreEqual(storedLinkValue.Source, value) ||
446         AreEqual(storedLinkValue.Target, value))
447     {
448         return handler(GetLinkStruct(index));
449     }
450     return @continue;
451 }
452 }
453 throw new NotSupportedException("Другие размеры и способы ограничений не
454     ↪ поддерживаются.");
455 }
456
457 /// <remarks>
458 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
459     ↪ в другом месте (но не в менеджере памяти, а в логике Links)
460 /// </remarks>
461 [MethodImpl(MethodImplOptions.AggressiveInlining)]
462 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
463 {
464     var constants = Constants;
465     var @null = constants.Null;
466     var externalReferencesRange = constants.ExternalReferencesRange;
467     var linkIndex = restrictions[constants.IndexPart];
468     ref var link = ref GetLinkDataPartReference(linkIndex);
469     var source = link.Source;

```

```

468 var target = link.Target;
469 ref var header = ref GetHeaderReference();
470 ref var rootAsSource = ref header.RootAsSource;
471 ref var rootAsTarget = ref header.RootAsTarget;
472 // Будет корректно работать только в том случае, если пространство выделенной связи
↪ предварительно заполнено нулями
473 if (!AreEqual(source, @null))
474 {
475     if (externalReferencesRange.HasValue &&
↪ externalReferencesRange.Value.Contains(source))
476     {
477         ExternalSourcesTreeMethods.Detach(ref rootAsSource, linkIndex);
478     }
479     else
480     {
481         InternalSourcesTreeMethods.Detach(ref
↪ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
482     }
483 }
484 if (!AreEqual(target, @null))
485 {
486     if (externalReferencesRange.HasValue &&
↪ externalReferencesRange.Value.Contains(target))
487     {
488         ExternalTargetsTreeMethods.Detach(ref rootAsTarget, linkIndex);
489     }
490     else
491     {
492         InternalTargetsTreeMethods.Detach(ref
↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
493     }
494 }
495 source = link.Source = substitution[constants.SourcePart];
496 target = link.Target = substitution[constants.TargetPart];
497 if (!AreEqual(source, @null))
498 {
499     if (externalReferencesRange.HasValue &&
↪ externalReferencesRange.Value.Contains(source))
500     {
501         ExternalSourcesTreeMethods.Attach(ref rootAsSource, linkIndex);
502     }
503     else
504     {
505         InternalSourcesTreeMethods.Attach(ref
↪ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
506     }
507 }
508 if (!AreEqual(target, @null))
509 {
510     if (externalReferencesRange.HasValue &&
↪ externalReferencesRange.Value.Contains(target))
511     {
512         ExternalTargetsTreeMethods.Attach(ref rootAsTarget, linkIndex);
513     }
514     else
515     {
516         InternalTargetsTreeMethods.Attach(ref
↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
517     }
518 }
519 return linkIndex;
520 }
521
522 /// <remarks>
523 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
↪ пространство
524 /// </remarks>
525 [MethodImpl(MethodImplOptions.AggressiveInlining)]
526 public virtual TLink Create(ICollection<TLink> restrictions)
527 {
528     ref var header = ref GetHeaderReference();
529     var freeLink = header.FirstFreeLink;
530     if (!AreEqual(freeLink, Constants.Null))
531     {
532         UnusedLinksListMethods.Detach(freeLink);
533     }
534     else
535     {

```



```

536     var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
537     if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
538     {
539         throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
540     }
541     if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
542     {
543         _dataMemory.ReservedCapacity += _dataMemory.ReservationStepInBytes;
544         _indexMemory.ReservedCapacity += _indexMemory.ReservationStepInBytes;
545         SetPointers(_dataMemory, _indexMemory);
546         header = ref GetHeaderReference();
547         header.ReservedLinks = ConvertToAddress(_dataMemory.ReservedCapacity /
548             ↳ LinkDataPartSizeInBytes);
549     }
550     header.AllocatedLinks = Increment(header.AllocatedLinks);
551     _dataMemory.UsedCapacity += LinkDataPartSizeInBytes;
552     _indexMemory.UsedCapacity += LinkIndexPartSizeInBytes;
553     freeLink = header.AllocatedLinks;
554 }
555 return freeLink;
556 }
557 [MethodImpl(MethodImplOptions.AggressiveInlining)]
558 public virtual void Delete(IList<TLink> restrictions)
559 {
560     ref var header = ref GetHeaderReference();
561     var link = restrictions[Constants.IndexPart];
562     if (LessThan(link, header.AllocatedLinks))
563     {
564         UnusedLinksListMethods.AttachAsFirst(link);
565     }
566     else if (AreEqual(link, header.AllocatedLinks))
567     {
568         header.AllocatedLinks = Decrement(header.AllocatedLinks);
569         _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
570         _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
571         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
572         // ↳ пока не дойдём до первой существующей связи
573         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
574         while (GreaterThan(header.AllocatedLinks, GetZero()) &&
575             ↳ IsUnusedLink(header.AllocatedLinks))
576         {
577             UnusedLinksListMethods.Detach(header.AllocatedLinks);
578             header.AllocatedLinks = Decrement(header.AllocatedLinks);
579             _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
580             _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
581         }
582     }
583 }
584 [MethodImpl(MethodImplOptions.AggressiveInlining)]
585 public IList<TLink> GetLinkStruct(TLink linkIndex)
586 {
587     ref var link = ref GetLinkDataPartReference(linkIndex);
588     return new Link<TLink>(linkIndex, link.Source, link.Target);
589 }
590 /// <remarks>
591 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
592 /// ↳ адрес реально поменялся
593 ///
594 /// Указатель this.links может быть в том же месте,
595 /// так как 0-я связь не используется и имеет такой же размер как Header,
596 /// поэтому header размещается в том же месте, что и 0-я связь
597 /// </remarks>
598 [MethodImpl(MethodImplOptions.AggressiveInlining)]
599 protected abstract void SetPointers(IResizableDirectMemory dataMemory,
600     ↳ IResizableDirectMemory indexMemory);
601 [MethodImpl(MethodImplOptions.AggressiveInlining)]
602 protected virtual void ResetPointers()
603 {
604     InternalSourcesTreeMethods = null;
605     ExternalSourcesTreeMethods = null;
606     InternalTargetsTreeMethods = null;
607     ExternalTargetsTreeMethods = null;
608     UnusedLinksListMethods = null;
609 }

```

```

610 [MethodImpl(MethodImplOptions.AggressiveInlining)]
611 protected abstract ref LinksHeader<TLink> GetHeaderReference();
612
613 [MethodImpl(MethodImplOptions.AggressiveInlining)]
614 protected abstract ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex);
615
616 [MethodImpl(MethodImplOptions.AggressiveInlining)]
617 protected abstract ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
    ↪ linkIndex);
618
619 [MethodImpl(MethodImplOptions.AggressiveInlining)]
620 protected virtual bool Exists(TLink link)
621     => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
622     && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
623     && !IsUnusedLink(link);
624
625 [MethodImpl(MethodImplOptions.AggressiveInlining)]
626 protected virtual bool IsUnusedLink(TLink linkIndex)
627 {
628     if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
        ↪ is not needed
        {
629         // TODO: Reduce access to memory in different location (should be enough to use
        ↪ just linkIndexPart)
630         ref var linkDataPart = ref GetLinkDataPartReference(linkIndex);
631         ref var linkIndexPart = ref GetLinkIndexPartReference(linkIndex);
632         return AreEqual(linkIndexPart.SizeAsSource, default) &&
        ↪ !AreEqual(linkDataPart.Source, default);
633     }
634     else
635     {
636         return true;
637     }
638 }
639
640
641 [MethodImpl(MethodImplOptions.AggressiveInlining)]
642 protected virtual TLink GetOne() => _one;
643
644 [MethodImpl(MethodImplOptions.AggressiveInlining)]
645 protected virtual TLink GetZero() => default;
646
647 [MethodImpl(MethodImplOptions.AggressiveInlining)]
648 protected virtual bool AreEqual(TLink first, TLink second) =>
    ↪ _equalityComparer.Equals(first, second);
649
650 [MethodImpl(MethodImplOptions.AggressiveInlining)]
651 protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
    ↪ second) < 0;
652
653 [MethodImpl(MethodImplOptions.AggressiveInlining)]
654 protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
    ↪ _comparer.Compare(first, second) <= 0;
655
656 [MethodImpl(MethodImplOptions.AggressiveInlining)]
657 protected virtual bool GreaterThan(TLink first, TLink second) =>
    ↪ _comparer.Compare(first, second) > 0;
658
659 [MethodImpl(MethodImplOptions.AggressiveInlining)]
660 protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
    ↪ _comparer.Compare(first, second) >= 0;
661
662 [MethodImpl(MethodImplOptions.AggressiveInlining)]
663 protected virtual long ConvertToInt64(TLink value) =>
    ↪ _addressToInt64Converter.Convert(value);
664
665 [MethodImpl(MethodImplOptions.AggressiveInlining)]
666 protected virtual TLink ConvertToAddress(long value) =>
    ↪ _int64ToAddressConverter.Convert(value);
667
668 [MethodImpl(MethodImplOptions.AggressiveInlining)]
669 protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
    ↪ second);
670
671 [MethodImpl(MethodImplOptions.AggressiveInlining)]
672 protected virtual TLink Subtract(TLink first, TLink second) =>
    ↪ Arithmetic<TLink>.Subtract(first, second);
673
674 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

675     protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
676
677     [MethodImpl(MethodImplOptions.AggressiveInlining)]
678     protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
679
680     #region Disposable
681
682     protected override bool AllowMultipleDisposeCalls
683     {
684         [MethodImpl(MethodImplOptions.AggressiveInlining)]
685         get => true;
686     }
687
688     [MethodImpl(MethodImplOptions.AggressiveInlining)]
689     protected override void Dispose(bool manual, bool wasDisposed)
690     {
691         if (!wasDisposed)
692         {
693             ResetPointers();
694             _dataMemory.DisposeIfPossible();
695             _indexMemory.DisposeIfPossible();
696         }
697     }
698
699     #endregion
700 }
701 }

```

1.36 ./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 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.Memory.Split.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 RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
32             ↳ AsRef<RawLinkDataPart<TLink>>(_links + RawLinkDataPart<TLink>.SizeInBytes *
33             ↳ _addressToInt64Converter.Convert(link));
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected override TLink GetPrevious(TLink element) =>
43             ↳ GetLinkDataPartReference(element).Source;
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override TLink GetNext(TLink element) =>
47             ↳ GetLinkDataPartReference(element).Target;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override TLink GetSize() => GetHeaderReference().FreeLinks;
51
52     }
53 }

```

```

45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
        ↪ element;
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
        ↪ element;
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override void SetPrevious(TLink element, TLink previous) =>
        ↪ GetLinkDataPartReference(element).Source = previous;
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected override void SetNext(TLink element, TLink next) =>
        ↪ GetLinkDataPartReference(element).Target = next;
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
59 }
60 }

```

1.37 ./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 =
            ↪ EqualityComparer<TLink>.Default;
13
14         public static readonly long SizeInBytes = Structure<RawLinkDataPart<TLink>>.Size;
15
16         public TLink Source;
17         public TLink Target;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public override bool Equals(object obj) => obj is RawLinkDataPart<TLink> link ?
            ↪ Equals(link) : false;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public bool Equals(RawLinkDataPart<TLink> other)
24             => _equalityComparer.Equals(Source, other.Source)
25             && _equalityComparer.Equals(Target, other.Target);
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public override int GetHashCode() => (Source, Target).GetHashCode();
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public static bool operator ==(RawLinkDataPart<TLink> left, RawLinkDataPart<TLink>
            ↪ right) => left.Equals(right);
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public static bool operator !=(RawLinkDataPart<TLink> left, RawLinkDataPart<TLink>
            ↪ right) => !(left == right);
35     }
36 }

```

1.38 ./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 =
            ↪ EqualityComparer<TLink>.Default;
13
14         public static readonly long SizeInBytes = Structure<RawLinkIndexPart<TLink>>.Size;
15

```

```

16     public TLink RootAsSource;
17     public TLink LeftAsSource;
18     public TLink RightAsSource;
19     public TLink SizeAsSource;
20     public TLink RootAsTarget;
21     public TLink LeftAsTarget;
22     public TLink RightAsTarget;
23     public TLink SizeAsTarget;
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     public override bool Equals(object obj) => obj is RawLinkIndexPart<TLink> link ?
        ↳ Equals(link) : false;
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     public bool Equals(RawLinkIndexPart<TLink> other)
30     => _equalityComparer.Equals(RootAsSource, other.RootAsSource)
31     && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
32     && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
33     && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
34     && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
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() => (RootAsSource, LeftAsSource, RightAsSource,
        ↳ SizeAsSource, RootAsTarget, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     public static bool operator ==(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
        ↳ right) => left.Equals(right);
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     public static bool operator !=(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
        ↳ right) => !(left == right);
47 }
48 }

```

1.39 ./csharp/Platform.Data.Doublets/Memory/United/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.Memory.United.Generic
13 {
14     public unsafe abstract class LinksAvlBalancedTreeMethodsBase<TLink> :
        ↳ SizedAndThreadedAVLBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
15     {
16         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            ↳ UncheckedConverter<TLink, long>.Default;
17         private static readonly UncheckedConverter<TLink, int> _addressToInt32Converter =
            ↳ UncheckedConverter<TLink, int>.Default;
18         private static readonly UncheckedConverter<bool, TLink> _boolToAddressConverter =
            ↳ UncheckedConverter<bool, TLink>.Default;
19         private static readonly UncheckedConverter<TLink, bool> _addressToBoolConverter =
            ↳ UncheckedConverter<TLink, bool>.Default;
20         private static readonly UncheckedConverter<int, TLink> _int32ToAddressConverter =
            ↳ UncheckedConverter<int, TLink>.Default;
21
22         protected readonly TLink Break;
23         protected readonly TLink Continue;
24         protected readonly byte* Links;
25         protected readonly byte* Header;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
            ↳ byte* header)
29         {
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,
    ↪ _boolToAddressConverter.Convert(value), 4, 1);
100         storedValue = modified;
101     }
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             }
178             if (AreEqual(index, leftSize))
179             {
180                 return root;
181             }
182             root = GetRightOrDefault(root);
183         }
184     }
185 }

```

```

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             root = GetRightOrDefault(root);
249         }
250     }
251     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
252 }

```



```

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 }

```

1.40 ./csharp/Platform.Data.Doublets/Memory/United/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.United.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;

```

```

19     protected readonly byte* Links;
20     protected readonly byte* Header;
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
24     ↪ byte* header)
25     {
26         Links = links;
27         Header = header;
28         Break = constants.Break;
29         Continue = constants.Continue;
30     }
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     protected abstract TLink GetTreeRoot();
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected abstract TLink GetBasePartValue(TLink link);
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
40     ↪ rootSource, TLink rootTarget);
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
44     ↪ rootSource, TLink rootTarget);
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
48     ↪ AsRef<LinksHeader<TLink>>(Header);
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
52     ↪ AsRef<RawLink<TLink>>(Links + RawLink<TLink>.SizeInBytes *
53     ↪ _addressToInt64Converter.Convert(link));
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
57     {
58         ref var link = ref GetLinkReference(linkIndex);
59         return new Link<TLink>(linkIndex, link.Source, link.Target);
60     }
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
64     {
65         ref var firstLink = ref GetLinkReference(first);
66         ref var secondLink = ref GetLinkReference(second);
67         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
68         ↪ secondLink.Source, secondLink.Target);
69     }
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
73     {
74         ref var firstLink = ref GetLinkReference(first);
75         ref var secondLink = ref GetLinkReference(second);
76         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
77         ↪ secondLink.Source, secondLink.Target);
78     }
79
80     public TLink this[TLink index]
81     {
82         [MethodImpl(MethodImplOptions.AggressiveInlining)]
83         get
84         {
85             var root = GetTreeRoot();
86             if (GreaterOrEqualThan(index, GetSize(root)))
87             {
88                 return Zero;
89             }
90             while (!EqualToZero(root))
91             {
92                 var left = GetLeftOrDefault(root);
93                 var leftSize = GetSizeOrZero(left);
94                 if (LessThan(index, leftSize))
95                 {
96                     root = left;
97                     continue;
98                 }
99             }
100         }
101     }

```

```

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

```

1.41 ./csharp/Platform.Data.Doublets/Memory/United/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.Memory.United.Generic
6 {
7     public unsafe class LinksSourcesAvlBalancedTreeMethods<TLink> :
8         ↳ LinksAvlBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

```

71         link.RightAsSource = Zero;
72         link.SizeAsSource = Zero;
73     }
74 }
75 }

```

1.42 ./csharp/Platform.Data.Doublets/Memory/United/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.United.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().RootAsSource;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool FirstIsToTheLeftOfSecond(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 }

```

1.43 ./csharp/Platform.Data.Doublets/Memory/United/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.Memory.United.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().RootAsTarget;
70
71         [MethodImpl(MethodImplOptions.AggressiveInlining)]
72         protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
73
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]
75         protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
76             ↳ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
77             ↳ AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource);
78
79         [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     }
81 }

```

```

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 }

```

1.44 ./csharp/Platform.Data.Doublets/Memory/United/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.United.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().RootAsTarget;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override bool FirstIsToTheLeftOfSecond(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 }

```

1.45 ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinks.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.United.Generic
10 {
11     public unsafe class UnitedMemoryLinks<TLink> : UnitedMemoryLinksBase<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 UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
20
21         /// <summary>
22         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
23         /// → минимальным шагом расширения базы данных.
24         /// </summary>
25         /// <param name="address">Полный путь к файлу базы данных.</param>
26         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
27         /// → байтах.</param>
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
30         → FileMappedResizableDirectMemory(address, memoryReservationStep),
31         → memoryReservationStep) { }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
35         → DefaultLinksSizeStep) { }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep) :
39         → this(memory, memoryReservationStep, Default<LinksConstants<TLink>>.Instance, true) {
40         → }
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep,
44         → LinksConstants<TLink> constants, bool useAvlBasedIndex) : base(memory,
45         → memoryReservationStep, constants)
46         {
47             if (useAvlBasedIndex)
48             {
49                 _createSourceTreeMethods = () => new
50                 → LinksSourcesAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
51                 _createTargetTreeMethods = () => new
52                 → LinksTargetsAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
53             }
54             else
55             {
56                 _createSourceTreeMethods = () => new
57                 → LinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
58                 _createTargetTreeMethods = () => new
59                 → LinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
60             }
61             Init(memory, memoryReservationStep);
62         }
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected override void SetPointers(IResizableDirectMemory memory)
66         {
67             _links = (byte*)memory.Pointer;
68             _header = _links;
69             SourcesTreeMethods = _createSourceTreeMethods();
70             TargetsTreeMethods = _createTargetTreeMethods();
71             UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_links, _header);
72         }
73     }
74 }

```

```

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));

```

1.46 ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinksBase.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.United.Generic
14 {
15     public abstract class UnitedMemoryLinksBase<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
56             {
57                 ref var header = ref GetHeaderReference();
58                 return Subtract(header.AllocatedLinks, header.FreeLinks);
59             }
60         }
61     }
62
63 }

```

```

57 public virtual LinksConstants<TLink> Constants
58 {
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     get;
61 }
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected UnitedMemoryLinksBase(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 UnitedMemoryLinksBase(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) ||
            ↪ AreEqual(storedLinkValue.Target, value))
131         {
132             return GetOne();
133         }
134         return GetZero();
135     }
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) &&
                ↪ AreEqual(storedLinkValue.Target, target))
176             {
177                 return GetOne();
178             }
179             return GetZero();
180         }
181         var value = default(TLink);
182         if (AreEqual(source, any))
183         {
184             value = target;
185         }
186         if (AreEqual(target, any))
187         {
188             value = source;
189         }
190         if (AreEqual(storedLinkValue.Source, value) ||
            ↪ AreEqual(storedLinkValue.Target, value))
191         {
192             return GetOne();
193         }
194         return GetZero();
195     }
196 }
197 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↪ поддерживаются.");
198 }
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         {
274             return Each(handler, Array.Empty<TLink>());
275         }
276         else if (AreEqual(source, any))
277         {
278             return TargetsTreeMethods.EachUsage(target, handler);
279         }
280         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
    ↪ в другом месте (но не в менеджере памяти, а в логике Links)
331 /// </remarks>
332 [MethodImpl(MethodImplOptions.AggressiveInlining)]
333 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
334 {
335     var constants = Constants;
336     var @null = constants.Null;
337     var linkIndex = restrictions[constants.IndexPart];
338     ref var link = ref GetLinkReference(linkIndex);
339     ref var header = ref GetHeaderReference();
340     ref var firstAsSource = ref header.RootAsSource;
341     ref var firstAsTarget = ref header.RootAsTarget;
342     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↪ предварительно заполнено нулями
343     if (!AreEqual(link.Source, @null))
344     {
345         SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
346     }
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(IList<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(IList<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)]
426 public IList<TLink> GetLinkStruct(TLink linkIndex)
427 {
428     ref var link = ref GetLinkReference(linkIndex);
429     return new Link<TLink>(linkIndex, link.Source, link.Target);
430 }
431
432 /// <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)]
501  protected virtual long ConvertToInt64(TLink value) =>
502  ↪ _addressToInt64Converter.Convert(value);
503
504  [MethodImpl(MethodImplOptions.AggressiveInlining)]
505  protected virtual TLink ConvertToAddress(long value) =>
506  ↪ _int64ToAddressConverter.Convert(value);

```



```

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

```

1.47 ./csharp/Platform.Data.Doublets/Memory/United/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.Memory.United.Generic
9  {
10     public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
    ↪     ILinksListMethods<TLink>
11     {
12         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
    ↪     UncheckedConverter<TLink, long>.Default;
13
14         private readonly byte* _links;
15         private readonly byte* _header;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public UnusedLinksListMethods(byte* links, byte* header)
19         {
20             _links = links;
21             _header = header;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
    ↪     AsRef<LinksHeader<TLink>>(_header);
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
    ↪     AsRef<RawLink<TLink>>(_links + RawLink<TLink>.SizeInBytes *
    ↪     _addressToInt64Converter.Convert(link));
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override TLink GetPrevious(TLink element) => GetLinkReference(element).Source;
38

```

```

39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected override TLink GetNext(TLink element) => GetLinkReference(element).Target;
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override TLink GetSize() => GetHeaderReference().FreeLinks;
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
47         ↪ element;
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
51         ↪ element;
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override void SetPrevious(TLink element, TLink previous) =>
55         ↪ GetLinkReference(element).Source = previous;
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override void SetNext(TLink element, TLink next) =>
59         ↪ GetLinkReference(element).Target = next;
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
63 }
64 }

```

1.48 ./csharp/Platform.Data.Doublets/Memory/United/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.Memory.United
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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public override bool Equals(object obj) => obj is RawLink<TLink> link ? Equals(link) :
28             ↪ false;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public bool Equals(RawLink<TLink> other)
32             => _equalityComparer.Equals(Source, other.Source)
33             && _equalityComparer.Equals(Target, other.Target)
34             && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
35             && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
36             && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
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() => (Source, Target, LeftAsSource, RightAsSource,
43             ↪ SizeAsSource, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public static bool operator ==(RawLink<TLink> left, RawLink<TLink> right) =>
47             ↪ left.Equals(right);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public static bool operator !=(RawLink<TLink> left, RawLink<TLink> right) => !(left ==
51             ↪ right);
52     }
53 }

```

```
48 }
```

1.49 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs

```
1 using System.Runtime.CompilerServices;
2 using Platform.Data.Doublets.Memory.United.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.Memory.United.Specific
8 {
9     public unsafe abstract class UInt64LinksAvlBalancedTreeMethodsBase :
10         ↳ LinksAvlBalancedTreeMethodsBase<ulong>
11     {
12         protected new readonly RawLink<ulong>* Links;
13         protected new readonly LinksHeader<ulong>* Header;
14
15         protected UInt64LinksAvlBalancedTreeMethodsBase(LinksConstants<ulong> constants,
16             ↳ RawLink<ulong>* links, LinksHeader<ulong>* header)
17             : base(constants, (byte*)links, (byte*)header)
18         {
19             Links = links;
20             Header = header;
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override ulong GetZero() => OUL;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override bool EqualToZero(ulong value) => value == OUL;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override bool AreEqual(ulong first, ulong second) => first == second;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override bool GreaterThanZero(ulong value) => value > OUL;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override bool GreaterThan(ulong first, ulong second) => first > second;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
43             ↳ always true for ulong
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override bool LessOrEqualThanZero(ulong value) => value == OUL; // value is
47             ↳ always >= 0 for ulong
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
54             ↳ for ulong
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override bool LessThan(ulong first, ulong second) => first < second;
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override ulong Increment(ulong value) => ++value;
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         protected override ulong Decrement(ulong value) => --value;
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         protected override ulong Add(ulong first, ulong second) => first + second;
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         protected override ulong Subtract(ulong first, ulong second) => first - second;
70
71         [MethodImpl(MethodImplOptions.AggressiveInlining)]
72         protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
73         {
74             ref var firstLink = ref Links[first];
75             ref var secondLink = ref Links[second];
76             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
77                 ↳ secondLink.Source, secondLink.Target);
78         }
79     }
80 }
```

```

72     }
73
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
76     {
77         ref var firstLink = ref Links[first];
78         ref var secondLink = ref Links[second];
79         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
80             ↪ secondLink.Source, secondLink.Target);
81     }
82
83     [MethodImpl(MethodImplOptions.AggressiveInlining)]
84     protected override ulong GetSizeValue(ulong value) => (value & 4294967264UL) >> 5;
85
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
88         ↪ storedValue & 31UL | (size & 134217727UL) << 5;
89
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     protected override bool GetLeftIsChildValue(ulong value) => (value & 16UL) >> 4 == 1UL;
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
95         ↪ storedValue = storedValue & 4294967279UL | (As<bool, byte>(ref value) & 1UL) << 4;
96
97     [MethodImpl(MethodImplOptions.AggressiveInlining)]
98     protected override bool GetRightIsChildValue(ulong value) => (value & 8UL) >> 3 == 1UL;
99
100    [MethodImpl(MethodImplOptions.AggressiveInlining)]
101    protected override sbyte GetBalanceValue(ulong value) => unchecked((sbyte)(value & 7UL |
102        ↪ 0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of
103        ↪ sbyte
104
105    [MethodImpl(MethodImplOptions.AggressiveInlining)]
106    protected override void SetBalanceValue(ref ulong storedValue, sbyte value) =>
107        ↪ storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
108        ↪ value & 3) & 7UL);
109
110    [MethodImpl(MethodImplOptions.AggressiveInlining)]
111    protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
112
113    [MethodImpl(MethodImplOptions.AggressiveInlining)]
114    protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
115 }

```

1.50 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs

```

1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.United.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.United.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
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          protected override ulong GetZero() => 0UL;
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          protected override bool EqualToZero(ulong value) => value == 0UL;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

27     protected override bool AreEqual(ulong first, ulong second) => first == second;
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     protected override bool GreaterThanZero(ulong value) => value > 0UL;
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     protected override bool GreaterThan(ulong first, ulong second) => first > second;
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
    ↪ always true for ulong
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
    ↪ always >= 0 for ulong
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
    ↪ for ulong
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override bool LessThan(ulong first, ulong second) => first < second;
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override ulong Increment(ulong value) => ++value;
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected override ulong Decrement(ulong value) => --value;
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override ulong Add(ulong first, ulong second) => first + second;
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected override ulong Subtract(ulong first, ulong second) => first - second;
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
67     {
68         ref var firstLink = ref Links[first];
69         ref var secondLink = ref Links[second];
70         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
71     }
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
75     {
76         ref var firstLink = ref Links[first];
77         ref var secondLink = ref Links[second];
78         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
79     }
80
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
83
84     [MethodImpl(MethodImplOptions.AggressiveInlining)]
85     protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
86 }
87 }

```

1.51 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs

```

1     using System.Runtime.CompilerServices;
2
3     #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5     namespace Platform.Data.Doublets.Memory.United.Specific
6     {
7         public unsafe class UInt64LinksSourcesAvlBalancedTreeMethods :
    ↪ UInt64LinksAvlBalancedTreeMethodsBase
8         {

```

```

9      public UInt64LinksSourcesAvlBalancedTreeMethods(LinksConstants<ulong> constants,
10         ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
11         ↳ { }
12
13     [MethodImpl(MethodImplOptions.AggressiveInlining)]
14     protected override ref ulong GetLeftReference(ulong node) => ref
15         ↳ Links[node].LeftAsSource;
16
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     protected override ref ulong GetRightReference(ulong node) => ref
19         ↳ Links[node].RightAsSource;
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
26         ↳ left;
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
30         ↳ right;
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsSource);
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
37         ↳ Links[node].SizeAsSource, size);
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected override bool GetLeftIsChild(ulong node) =>
41         ↳ GetLeftIsChildValue(Links[node].SizeAsSource);
42
43     // [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     // protected override bool GetLeftIsChild(ulong node) => IsChild(node, GetLeft(node));
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override void SetLeftIsChild(ulong node, bool value) =>
48         ↳ SetLeftIsChildValue(ref Links[node].SizeAsSource, value);
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override bool GetRightIsChild(ulong node) =>
52         ↳ GetRightIsChildValue(Links[node].SizeAsSource);
53
54     // [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     // protected override bool GetRightIsChild(ulong node) => IsChild(node, GetRight(node));
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override void SetRightIsChild(ulong node, bool value) =>
59         ↳ SetRightIsChildValue(ref Links[node].SizeAsSource, value);
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     protected override sbyte GetBalance(ulong node) =>
63         ↳ GetBalanceValue(Links[node].SizeAsSource);
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
67         ↳ Links[node].SizeAsSource, value);
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override ulong GetTreeRoot() => Header->RootAsSource;
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
74
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
77         ↳ ulong secondSource, ulong secondTarget)
78         => firstSource < secondSource || firstSource == secondSource && firstTarget <
79         ↳ secondTarget;
80
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
83         ↳ ulong secondSource, ulong secondTarget)

```

```

71         => firstSource > secondSource || firstSource == secondSource && firstTarget >
           ↳ secondTarget;
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     protected override void ClearNode(ulong node)
75     {
76         ref var link = ref Links[node];
77         link.LeftAsSource = OUL;
78         link.RightAsSource = OUL;
79         link.SizeAsSource = OUL;
80     }
81 }
82 }

```

1.52 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.United.Specific
6  {
7      public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
8          ↳ UInt64LinksSizeBalancedTreeMethodsBase
9      {
10
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         protected override ref ulong GetLeftReference(ulong node) => ref
13             ↳ Links[node].LeftAsSource;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected override ref ulong GetRightReference(ulong node) => ref
17             ↳ Links[node].RightAsSource;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
27             ↳ left;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
31             ↳ right;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override ulong GetSize(ulong node) => Links[node].SizeAsSource;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsSource =
38             ↳ size;
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override ulong GetTreeRoot() => Header->RootAsSource;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
48             ↳ ulong secondSource, ulong secondTarget)
49             => firstSource < secondSource || firstSource == secondSource && firstTarget <
50                 ↳ secondTarget;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
54             ↳ ulong secondSource, ulong secondTarget)
55             => firstSource > secondSource || firstSource == secondSource && firstTarget >
56                 ↳ secondTarget;
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         protected override void ClearNode(ulong node)
60         {
61             ref var link = ref Links[node];

```

```

53         link.LeftAsSource = OUL;
54         link.RightAsSource = OUL;
55         link.SizeAsSource = OUL;
56     }
57 }
58 }

```

1.53 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs

```

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

```



```

58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
60     ↪     ulong secondSource, ulong secondTarget)
61     => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
        ↪     secondSource;

62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
64     ↪     ulong secondSource, ulong secondTarget)
65     => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
        ↪     secondSource;

66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override void ClearNode(ulong node)
68     {
69         ref var link = ref Links[node];
70         link.LeftAsTarget = OUL;
71         link.RightAsTarget = OUL;
72         link.SizeAsTarget = OUL;
73     }
74 }
75 }
76 }

```

1.54 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.United.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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected override ref ulong GetLeftReference(ulong node) => ref
15         ↪     Links[node].LeftAsTarget;

16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected override ref ulong GetRightReference(ulong node) => ref
18         ↪     Links[node].RightAsTarget;

19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;

21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;

23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
25         ↪     left;

26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
28         ↪     right;

29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;

31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
33         ↪     size;

34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected override ulong GetTreeRoot() => Header->RootAsTarget;

36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override ulong GetBasePartValue(ulong link) => Links[link].Target;

38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
40         ↪     ulong secondSource, ulong secondTarget)
41         => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
42         ↪     secondSource;
43
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 }

```

1.55 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnitedMemoryLinks.cs

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Memory;
4  using Platform.Singletons;
5  using Platform.Data.Doublets.Memory.United.Generic;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Memory.United.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="ulong"
    ↪     />.</para>
13     /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
    ↪     размером, для организации хранения связей с адресами представленными в виде <see
    ↪     cref="ulong"/>.</para>
14     /// </summary>
15     public unsafe class UInt64UnitedMemoryLinks : UnitedMemoryLinksBase<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 UInt64UnitedMemoryLinks(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 UInt64UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
    ↪     FileMappedResizableDirectMemory(address, memoryReservationStep),
    ↪     memoryReservationStep) { }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public UInt64UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
    ↪     DefaultLinksSizeStep) { }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
    ↪     memoryReservationStep) : this(memory, memoryReservationStep,
    ↪     Default<LinksConstants<ulong>>.Instance, true) { }
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public UInt64UnitedMemoryLinks(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
50             ↳ UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
51         _createTargetTreeMethods = () => new
52             ↳ UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
53     }
54     Init(memory, memoryReservationStep);
55 }
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 protected override void SetPointers(IResizableDirectMemory memory)
58 {
59     _header = (LinksHeader<ulong>*)memory.Pointer;
60     _links = (RawLink<ulong>*)memory.Pointer;
61     SourcesTreeMethods = _createSourceTreeMethods();
62     TargetsTreeMethods = _createTargetTreeMethods();
63     UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
64 }
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override void ResetPointers()
67 {
68     base.ResetPointers();
69     _links = null;
70     _header = null;
71 }
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
76     ↳ _links[linkIndex];
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected override bool AreEqual(ulong first, ulong second) => first == second;
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 protected override bool LessThan(ulong first, ulong second) => first < second;
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 protected override bool GreaterThan(ulong first, ulong second) => first > second;
85 [MethodImpl(MethodImplOptions.AggressiveInlining)]
86 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 protected override ulong GetZero() => 0UL;
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 protected override ulong GetOne() => 1UL;
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 protected override long ConvertToInt64(ulong value) => (long)value;
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 protected override ulong ConvertToAddress(long value) => (ulong)value;
95 [MethodImpl(MethodImplOptions.AggressiveInlining)]
96 protected override ulong Add(ulong first, ulong second) => first + second;
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 protected override ulong Subtract(ulong first, ulong second) => first - second;
99 [MethodImpl(MethodImplOptions.AggressiveInlining)]
100 protected override ulong Increment(ulong link) => ++link;
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 protected override ulong Decrement(ulong link) => --link;
103 }
104 }
105 }
106 }
107 }
108 }
109 }
110 }
111 }
112 }
113 }
114 }
115 }
116 }
117 }
118 }

```

1.56 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnusedLinksListMethods.cs

```

1 using System.Runtime.CompilerServices;
2 using Platform.Data.Doublets.Memory.United.Generic;

```

```

3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Memory.United.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 }

```

1.57 ./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 }

```

1.58 ./csharp/Platform.Data.Doublets/Numbers/Unary/LinkToltsFrequencyNumberConveter.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)
25             : base(links)
26         {
27             _frequencyPropertyOperator = frequencyPropertyOperator;
28             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public TLink Convert(Doublet<TLink> doublet)
33         {
34             var links = _links;
35             var link = links.SearchOrDefault(doublet.Source, doublet.Target);
36             if (_equalityComparer.Equals(link, default))
37             {
38                 throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
39             }
40             var frequency = _frequencyPropertyOperator.Get(link);
41             if (_equalityComparer.Equals(frequency, default))
42             {
43                 return default;
44             }
45             var frequencyNumber = links.GetSource(frequency);
46             return _unaryNumberToAddressConverter.Convert(frequencyNumber);
47         }
48     }
49 }

```

1.59 ./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         }
36     }
37 }

```

```

32     var previousPowerOf2 = Convert(power - 1);
33     var powerOf2 = _links.GetOrCreate(previousPowerOf2, previousPowerOf2);
34     _unaryNumberPowersOf2[power] = powerOf2;
35     return powerOf2;
36 }
37 }
38 }

```

1.60 ./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>,
11         ⇨ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ⇨ EqualityComparer<TLink>.Default;
15         private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
16             ⇨ UncheckedConverter<TLink, ulong>.Default;
17         private static readonly UncheckedConverter<ulong, TLink> _uint64ToAddressConverter =
18             ⇨ UncheckedConverter<ulong, TLink>.Default;
19         private static readonly TLink _zero = default;
20         private static readonly TLink _one = Arithmetic.Increment(_zero);
21
22         private readonly Dictionary<TLink, TLink> _unaryToUInt64;
23         private readonly TLink _unaryOne;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
27             : base(links)
28         {
29             _unaryOne = unaryOne;
30             _unaryToUInt64 = CreateUnaryToUInt64Dictionary(links, unaryOne);
31         }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public TLink Convert(TLink unaryNumber)
35         {
36             if (_equalityComparer.Equals(unaryNumber, default))
37             {
38                 return default;
39             }
40             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
41             {
42                 return _one;
43             }
44             var links = _links;
45             var source = links.GetSource(unaryNumber);
46             var target = links.GetTarget(unaryNumber);
47             if (_equalityComparer.Equals(source, target))
48             {
49                 return _unaryToUInt64[unaryNumber];
50             }
51             else
52             {
53                 var result = _unaryToUInt64[source];
54                 TLink lastValue;
55                 while (!_unaryToUInt64.TryGetValue(target, out lastValue))
56                 {
57                     source = links.GetSource(target);
58                     result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
59                     target = links.GetTarget(target);
60                 }
61                 result = Arithmetic<TLink>.Add(result, lastValue);
62                 return result;
63             }
64         }
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         private static Dictionary<TLink, TLink> CreateUnaryToUInt64Dictionary(ILinks<TLink>
68             ⇨ links, TLink unaryOne)
69         {
70             var unaryToUInt64 = new Dictionary<TLink, TLink>
71             {

```

```

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) =>
82     ↪ _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
83 }

```

1.61 ./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             {

```

```

55         unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
56     }
57     return unaryNumberPowerOf2Indicies;
58 }
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 private static void SetBit(ref TLink target, int powerOf2Index) => target =
    ↪ Bit.Or(target, Bit.ShiftLeft(_one, powerOf2Index));
62 }
63 }

```

1.62 ./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 }

```

1.63 ./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 }

```

1.64 ./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     // 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(IList<TLink> destination, IList<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<IList<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 }

```

```

34
35 private struct HalfDoublet
36 {
37     public TLink Element;
38     public LinkFrequency<TLink> DoubletData;
39
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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
    ↳ doInitialFrequenciesIncrement)
    : this(links, baseConverter, doubletFrequenciesCache, _one,
    ↳ doInitialFrequenciesIncrement) { }
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
    ↳ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
    : base(links)
58 {
59     _baseConverter = baseConverter;
60     _doubletFrequenciesCache = doubletFrequenciesCache;
61     if (_comparer.Compare(minFrequencyToCompress, _one) < 0)
62     {
63         minFrequencyToCompress = _one;
64     }
65     _minFrequencyToCompress = minFrequencyToCompress;
66     _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
67     ResetMaxDoublet();
68 }
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 public override TLink Convert(IList<TLink> source) =>
72     ↳ _baseConverter.Convert(Compress(source));
73
74 /// <remarks>
75 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
76 /// Faster version (doublets' frequencies dictionary is not recreated).
77 /// </remarks>
78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 private IList<TLink> Compress(IList<TLink> sequence)
80 {
81     if (sequence.IsNullOrEmpty())
82     {
83         return null;
84     }
85     if (sequence.Count == 1)
86     {
87         return sequence;
88     }
89     if (sequence.Count == 2)
90     {
91         return new[] { _links.GetOrCreate(sequence[0], sequence[1]) };
92     }
93     // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
94     var copy = new HalfDoublet[sequence.Count];
95     Doublet<TLink> doublet = default;
96     for (var i = 1; i < sequence.Count; i++)
97     {
98         doublet.Source = sequence[i - 1];
99         doublet.Target = sequence[i];
100         LinkFrequency<TLink> data;
101         if (_doInitialFrequenciesIncrement)
102         {
103             data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
104         }
105

```

```

106     }
107     else
108     {
109         data = _doubletFrequenciesCache.GetFrequency(ref doublet);
110         if (data == null)
111         {
112             throw new NotSupportedException("If you ask not to increment
113                 ↪ frequencies, it is expected that all frequencies for the sequence
114                 ↪ are prepared.");
115         }
116     }
117     copy[i - 1].Element = sequence[i - 1];
118     copy[i - 1].DoubletData = data;
119     UpdateMaxDoublet(ref doublet, data);
120 }
121 copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
122 copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
123 if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
124 {
125     var newLength = ReplaceDoublets(copy);
126     sequence = new TLink[newLength];
127     for (int i = 0; i < newLength; i++)
128     {
129         sequence[i] = copy[i].Element;
130     }
131 }
132 return sequence;
133 }
134
135 /// <remarks>
136 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
137 /// </remarks>
138 [MethodImpl(MethodImplOptions.AggressiveInlining)]
139 private int ReplaceDoublets(HalfDoublet[] copy)
140 {
141     var oldLength = copy.Length;
142     var newLength = copy.Length;
143     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
144     {
145         var maxDoubletSource = _maxDoublet.Source;
146         var maxDoubletTarget = _maxDoublet.Target;
147         if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
148         {
149             _maxDoubletData.Link = _links.GetOrCreate(maxDoubletSource,
150                 ↪ maxDoubletTarget);
151         }
152         var maxDoubletReplacementLink = _maxDoubletData.Link;
153         oldLength--;
154         var oldLengthMinusTwo = oldLength - 1;
155         // Substitute all usages
156         int w = 0, r = 0; // (r == read, w == write)
157         for (; r < oldLength; r++)
158         {
159             if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
160                 ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
161             {
162                 if (r > 0)
163                 {
164                     var previous = copy[w - 1].Element;
165                     copy[w - 1].DoubletData.DecrementFrequency();
166                     copy[w - 1].DoubletData =
167                         ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
168                         ↪ maxDoubletReplacementLink);
169                 }
170                 if (r < oldLengthMinusTwo)
171                 {
172                     var next = copy[r + 2].Element;
173                     copy[r + 1].DoubletData.DecrementFrequency();
174                     copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
175                         ↪ next);
176                 }
177                 copy[w++].Element = maxDoubletReplacementLink;
178                 r++;
179                 newLength--;
180             }
181             else
182             {
183                 copy[w] = copy[r];
184                 w++;
185             }
186         }
187     }
188     return newLength;
189 }

```

```

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 random
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             }
76             else
77             {
78                 currentLevel = levels[i];
79                 levelRepeat = 1;
80                 if (skipOnce)
81                 {

```

```

75         skipOnce = false;
76     }
77     else
78     {
79         sequence[w] = sequence[i - 1];
80         levels[w] = levels[i - 1];
81         previousLevel = levels[w];
82         w++;
83     }
84     if (i == length - 1)
85     {
86         sequence[w] = sequence[i];
87         levels[w] = levels[i];
88         w++;
89     }
90 }
91 }
92 length = w;
93 }
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>,
           ↪ ICriterionMatcher<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public bool IsMatched(TLink argument) => _links.IsPartialPoint(argument);
15     }
16 }

```

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 =
           ↪ EqualityComparer<TLink>.Default;
12
13         private readonly ILinks<TLink> _links;
14         private readonly TLink _sequenceMarkerLink;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
18         {
19             _links = links;
20             _sequenceMarkerLink = sequenceMarkerLink;
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public bool IsMatched(TLink sequenceCandidate)
25             => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
26             || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
           ↪ sequenceCandidate), _links.Constants.Null);
27     }
28 }

```

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>,
           ↪ ISequenceAppender<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪ EqualityComparer<TLink>.Default;
14
15         private readonly IStack<TLink> _stack;
16         private readonly ISequenceHeightProvider<TLink> _heightProvider;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
           ↪ ISequenceHeightProvider<TLink> heightProvider)
20             : base(links)

```



```

21     {
22         _stack = stack;
23         _heightProvider = heightProvider;
24     }
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     public TLink Append(TLink sequence, TLink appendant)
28     {
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> :
    ↳ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
    ↳ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
19 {
20     private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
    ↳ UncheckedConverter<TLink, long>.Default;
21     private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
    ↳ UncheckedConverter<TLink, ulong>.Default;
22     private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
    ↳ UncheckedConverter<ulong, TLink>.Default;
23
24     private readonly IList<TLink> _links;
25     private readonly IList<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(IList<TLink> links, IList<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 }

```

```

48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 public void IncrementFrequencies(ICollection<TLink> sequence)
50 {
51     for (var i = 1; i < sequence.Count; i++)
52     {
53         IncrementFrequency(sequence[i - 1], sequence[i]);
54     }
55 }
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
59 {
60     var doublet = new Doublet<TLink>(source, target);
61     return IncrementFrequency(ref doublet);
62 }
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 public void PrintFrequencies(ICollection<TLink> sequence)
66 {
67     for (var i = 1; i < sequence.Count; i++)
68     {
69         PrintFrequency(sequence[i - 1], sequence[i]);
70     }
71 }
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 public void PrintFrequency(TLink source, TLink target)
75 {
76     var number = GetFrequency(source, target).Frequency;
77     Console.WriteLine("{0},{1} - {2}", source, target, number);
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
82 {
83     if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
84     {
85         data.IncrementFrequency();
86     }
87     else
88     {
89         var link = _links.SearchOrDefault(doublet.Source, doublet.Target);
90         data = new LinkFrequency<TLink>(_one, link);
91         if (!_equalityComparer.Equals(link, default))
92         {
93             data.Frequency = Arithmetic.Add(data.Frequency,
94                 ↪ _frequencyCounter.Count(link));
95         }
96         _doubletsCache.Add(doublet, data);
97     }
98     return data;
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             // TODO: 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             //if (value.Frequency > 0)
121             //    var frequency = value.Frequency;
122

```

```

123         //         linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
124         //         var count = _countLinkFrequency(linkIndex);
125
126         //         if ((frequency > count && frequency - count > 1) || (count > frequency
127         ↪ && count - frequency > 1))
128         //             throw new InvalidOperationException("Frequencies validation
129         ↪ failed.");
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/MarkedSequenceSymbolFrequencyOneOf

```

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> :
    ↪     SequenceSymbolFrequencyOneOffCounter<TLink>
9     {
10         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
    ↪         ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
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 =
    ↪         EqualityComparer<TLink>.Default;
14         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
15
16         protected readonly ILinks<TLink> _links;
17         protected readonly TLink _sequenceLink;
18         protected readonly TLink _symbol;
19         protected TLink _total;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
    ↪         TLink symbol)
23         {
24             _links = links;
25             _sequenceLink = sequenceLink;
26             _symbol = symbol;
27             _total = default;
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public virtual TLink Count()
32         {
33             if (_comparer.Compare(_total, default) > 0)
34             {
35                 return _total;
36             }
37             StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
    ↪             IsElement, VisitElement);
38             return _total;
39         }
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
    ↪         _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
    ↪         IsPartialPoint
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         private bool VisitElement(TLink element)
46         {
47             if (_equalityComparer.Equals(element, _symbol))
48             {
49                 _total = Arithmetic.Increment(_total);
50             }
51             return true;

```

```

52     }
53 }
54 }

```

1.79 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.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 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/TotalMarkedSequenceSymbolFrequencyOneOffCounter.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.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.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 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     }

```

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

```

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

```

73 }

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     }
54 }
```

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;
```

```

21         while (!_elementMatcher.IsMatched(pairOrElement))
22         {
23             pairOrElement = _links.GetTarget(pairOrElement);
24             height = Arithmetic.Increment(height);
25         }
26         return height;
27     }
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(ICollection<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(ICollection<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 }

```

```

55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     private bool IsIndexed(TLink source, TLink target)
57     {
58         var frequency = _cache.GetFrequency(source, target);
59         if (frequency == null)
60         {
61             return false;
62         }
63         return !_equalityComparer.Equals(frequency.Frequency, default);
64     }
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 }
63 }

```

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     private readonly ISynchronizedLinks<TLink> _links;
13
14     [MethodImpl(MethodImplOptions.AggressiveInlining)]
15     public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;
16
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     public bool Add(ICollection<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
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     public bool MightContain(ICollection<TLink> sequence)
43     {
44         var links = _links.Unsync;
45         return _links.SyncRoot.ExecuteReadOperation(() =>
46         {
47             var indexed = true;
48             var i = sequence.Count;
49             while (--i >= 1 && (indexed =
50                 ↪ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
51                 ↪ sequence[i]), default))) { }
52             return indexed;
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             [MethodImpl(MethodImplOptions.AggressiveInlining)]
88             public List<ulong> CreateAllVariants1(params ulong[] sequence)
89             {
90                 return _sync.ExecuteWriteOperation(() =>

```

```

91     {
92         return new List<ulong>();
93     }
94     Links.Unsync.EnsureLinkExists(sequence);
95     if (sequence.Length == 1)
96     {
97         return new List<ulong> { sequence[0] };
98     }
99     var results = new
100     ↪ List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
101     return CreateAllVariants1Core(sequence, results);
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
113             ↪ implemented.");
114         }
115         results.Add(link);
116         return results;
117     }
118     var innerSequenceLength = sequence.Length - 1;
119     var innerSequence = new ulong[innerSequenceLength];
120     for (var li = 0; li < innerSequenceLength; li++)
121     {
122         var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
123         if (link == Constants.Null)
124         {
125             throw new NotImplementedException("Creation cancellation is not
126             ↪ implemented.");
127         }
128         for (var isi = 0; isi < li; isi++)
129         {
130             innerSequence[isi] = sequence[isi];
131         }
132         innerSequence[li] = link;
133         for (var isi = li + 1; isi < innerSequenceLength; isi++)
134         {
135             innerSequence[isi] = sequence[isi + 1];
136         }
137         CreateAllVariants1Core(innerSequence, results);
138     }
139     return results;
140 }
141
142 #endregion
143
144 [MethodImpl(MethodImplOptions.AggressiveInlining)]
145 public HashSet<ulong> Each1(params ulong[] sequence)
146 {
147     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
148     Each1(link =>
149     {
150         if (!visitedLinks.Contains(link))
151         {
152             visitedLinks.Add(link); // изучить почему случаются повторы
153             return true;
154         }, sequence);
155     return visitedLinks;
156 }
157
158 [MethodImpl(MethodImplOptions.AggressiveInlining)]
159 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
160 {
161     if (sequence.Length == 2)
162     {
163         Links.Unsync.Each(sequence[0], sequence[1], handler);
164     }
165     else
166     {

```



```

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                 innerSequence[linkIndex] = doublet[Constants.IndexPart];
191                 Each1(handler, innerSequence);
192                 return Constants.Continue;
193             }, Constants.Any, left, right);
194         }
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.Unsync.Each(Constants.Any, Constants.Any, handler);
249     }
250 }
251 else if (sequence.Length == 2)
252 {
253     // _links.Each(sequence[0], sequence[1], handler);
254     //   o_|      x_o ...
255     // x_|      |__|
256     Links.Unsync.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.Unsync.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
464 [MethodImpl(MethodImplOptions.AggressiveInlining)]
465 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
466 {
467     return _sync.ExecuteReadOperation(() =>
468     {
469         var results = new HashSet<ulong>();
470         if (sequence.Length > 0)
471         {
472             Links.EnsureLinkExists(sequence);
473             var firstElement = sequence[0];
474             if (sequence.Length == 1)
475             {

```

```

474         results.Add(firstElement);
475         return results;
476     }
477     if (sequence.Length == 2)
478     {
479         var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
480         if (doublet != Constants.Null)
481         {
482             results.Add(doublet);
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             //if {
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         }
550         sb.Append(insertComma ? ", ..." : "...");
551     }
552     return false;

```

```

543         });
544     }
545     sb.Append('}');
546     return sb.ToString();
547 }
548
549 [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),
    ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
566     {
567         if (insertComma && sb.Length > 1)
568         {
569             sb.Append(',');
570         }
571         if (entered.Contains(element))
572         {
573             sb.Append('{');
574             elementToString(sb, element);
575             sb.Append('}');
576         }
577         else
578         {
579             elementToString(sb, element);
580         }
581         if (sb.Length < MaxSequenceFormatSize)
582         {
583             return true;
584         }
585         sb.Append(insertComma ? ", ..." : "...");
586         return false;
587     });
588     }
589     sb.Append('}');
590     return sb.ToString();
591 }
592
593 [MethodImpl(MethodImplOptions.AggressiveInlining)]
594 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
595 {
596     return _sync.ExecuteReadOperation(() =>
597     {
598         if (sequence.Length > 0)
599         {
600             Links.EnsureLinkExists(sequence);
601             var results = new HashSet<ulong>();
602             for (var i = 0; i < sequence.Length; i++)
603             {
604                 AllUsagesCore(sequence[i], results);
605             }
606             var filteredResults = new List<ulong>();
607             var linksInSequence = new HashSet<ulong>(sequence);
608             foreach (var result in results)
609             {
610                 var filterPosition = -1;
611                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
    ↪ Links.Unsync.GetTarget,

```

```

612         x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
613         ↪ x =>
614     {
615         if (filterPosition == (sequence.Length - 1))
616         {
617             return false;
618         }
619         if (filterPosition >= 0)
620         {
621             if (x == sequence[filterPosition + 1])
622             {
623                 filterPosition++;
624             }
625             else
626             {
627                 return false;
628             }
629         }
630         if (filterPosition < 0)
631         {
632             if (x == sequence[0])
633             {
634                 filterPosition = 0;
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         }
692     });
693 }

```

```

689         }
690         return true;
691     }
692     return true;
693 });
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             //{

```



```

766         // AllUsagesCore(sequence[i], nextResults);
767         // if (results.IsNullOrEmpty())
768         // {
769         //     results = nextResults;
770         //     nextResults = new HashSet<ulong>();
771         // }
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;

```

```

838     //});
839     //if (sequence.Length >= 2)
840     //    StepRight(handler, sequence[0], sequence[1]);
841     //var last = sequence.Length - 2;
842     //for (var i = 1; i < last; i++)
843     //    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     //    throw new NotImplementedException(); // all sequences, containing
849     //    ↪ this element?
850     //if (sequence.Length == 2)
851     //{
852     //    var results = new List<ulong>();
853     //    PartialStepRight(results.Add, sequence[0], sequence[1]);
854     //    return results;
855     //}
856     //var matches = new List<List<ulong>>();
857     //var last = sequence.Length - 1;
858     //for (var i = 0; i < last; i++)
859     //{
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     //}
880     //if (matches.Count > 0)
881     //{
882     //    var usages = new HashSet<ulong>();
883     //    for (int i = 0; i < sequence.Length; i++)
884     //    {
885     //        AllUsagesCore(sequence[i], usages);
886     //    }
887     //    //for (int i = 0; i < matches[0].Count; i++)
888     //    //    AllUsagesCore(matches[0][i], usages);
889     //    //usages.UnionWith(matches[0]);
890     //    return usages.ToList();
891     //}
892     var firstLinkUsages = new HashSet<ulong>();
893     AllUsagesCore(sequence[0], firstLinkUsages);
894     firstLinkUsages.Add(sequence[0]);
895     //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
896     //    ↪ sequence[0] }; // or all sequences, containing this element?
897     //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
898     //    ↪ 1).ToList();
899     var results = new HashSet<ulong>();
900     foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
901     ↪ firstLinkUsages, 1))
902     {
903         AllUsagesCore(match, results);
904     }
905     return results.ToList();
906 }
907 return new List<ulong>();
908 });
909 }
910
911 /// <remarks>
912 /// 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);

```

```

983         return counter.Count();
984     }
985 }
986
987 [MethodImpl(MethodImplOptions.AggressiveInlining)]
988 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
989     ↳ LinkIndex> outerHandler)
990 {
991     bool handler(ulong doublet)
992     {
993         if (usages.Add(doublet))
994         {
995             if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
996             {
997                 return false;
998             }
999             if (!AllUsagesCore1(doublet, usages, outerHandler))
1000             {
1001                 return false;
1002             }
1003         }
1004         return true;
1005     }
1006     return Links.Unsync.Each(link, Constants.Any, handler)
1007         && Links.Unsync.Each(Constants.Any, link, handler);
1008 }
1009
1010 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1011 public void CalculateAllUsages(ulong[] totals)
1012 {
1013     var calculator = new AllUsagesCalculator(Links, totals);
1014     calculator.Calculate();
1015 }
1016
1017 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1018 public void CalculateAllUsages2(ulong[] totals)
1019 {
1020     var calculator = new AllUsagesCalculator2(Links, totals);
1021     calculator.Calculate();
1022 }
1023
1024 private class AllUsagesCalculator
1025 {
1026     private readonly SynchronizedLinks<ulong> _links;
1027     private readonly ulong[] _totals;
1028
1029     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1030     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
1031     {
1032         _links = links;
1033         _totals = totals;
1034     }
1035
1036     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1037     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1038         ↳ CalculateCore);
1039
1040     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1041     private bool CalculateCore(ulong link)
1042     {
1043         if (_totals[link] == 0)
1044         {
1045             var total = 1UL;
1046             _totals[link] = total;
1047             var visitedChildren = new HashSet<ulong>();
1048             bool linkCalculator(ulong child)
1049             {
1050                 if (link != child && visitedChildren.Add(child))
1051                 {
1052                     total += _totals[child] == 0 ? 1 : _totals[child];
1053                 }
1054                 return true;
1055             }
1056             _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1057             _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1058             _totals[link] = total;
1059         }
1060         return true;
1061     }
1062 }

```

```

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

```

```

1137         {
1138             stack.Push(element);
1139             visitNode(element);
1140             element = getTarget(element);
1141         }
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(ICollection<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
1223 private class AllUsagesIntersectingCollector
1224 {
1225     private readonly SynchronizedLinks<ulong> _links;
1226     private readonly HashSet<ulong> _intersectWith;
1227     private readonly HashSet<ulong> _usages;
1228     private readonly HashSet<ulong> _enter;
1229
1230     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1231     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
        ↪ 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
        ↪ right)
1257 {
1258     TryStepLeftUp(handler, left, right);
1259     TryStepRightUp(handler, right, left);
1260 }
1261
1262 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1263 private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
        ↪ right)
1264 {
1265     // Direct
1266     if (left == right)
1267     {
1268         handler(new LinkAddress<LinkIndex>(left));
1269     }
1270     var doublet = Links.Unsync.SearchOrDefault(left, right);
1271     if (doublet != Constants.Null)
1272     {
1273         handler(new LinkAddress<LinkIndex>(doublet));
1274     }
1275     // Inner
1276     CloseInnerConnections(handler, left, right);
1277     // Outer
1278     StepLeft(handler, left, right);
1279     StepRight(handler, left, right);
1280     PartialStepRight(handler, left, right);
1281     PartialStepLeft(handler, left, right);
1282 }
1283
1284 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1285 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
        ↪ HashSet<ulong> previousMatchings, long startAt)
1286 {
1287     if (startAt >= sequence.Length) // ?
1288     {
1289         return previousMatchings;
1290     }
1291     var secondLinkUsages = new HashSet<ulong>();

```

```

1292 AllUsagesCore(sequence[startAt], secondLinkUsages);
1293 secondLinkUsages.Add(sequence[startAt]);
1294 var matchings = new HashSet<ulong>();
1295 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     });
1371 }

```



```

1359         return new HashSet<ulong>();
1360     });
1361 }
1362
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
1366 //        несколько раз в последовательности)
1367 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1368 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1369 {
1370     return _sync.ExecuteReadOperation(() =>
1371     {
1372         var results = new HashSet<ulong>();
1373         if (linksToConnect.Length > 0)
1374         {
1375             Links.EnsureLinkExists(linksToConnect);
1376             AllUsagesCore(linksToConnect[0], results);
1377             for (var i = 1; i < linksToConnect.Length; i++)
1378             {
1379                 var next = new HashSet<ulong>();
1380                 AllUsagesCore(linksToConnect[i], next);
1381                 results.IntersectWith(next);
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             return results;
1407         }
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             return results;
1433         }
1434     });
1435 }

```

```

1436 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1437 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1438 {
1439     return _sync.ExecuteReadOperation(() =>
1440     {
1441         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1442         ↪ BitArray((int)_links.Total + 1);
1443         if (linksToConnect.Length > 0)
1444         {
1445             Links.EnsureLinkExists(linksToConnect);
1446             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1447             collector1.Collect(linksToConnect[0]);
1448             for (var i = 1; i < linksToConnect.Length; i++)
1449             {
1450                 var next = new BitString((long)Links.Unsync.Count() + 1); //new
1451                 ↪ BitArray((int)_links.Total + 1);
1452                 var collector = new AllUsagesCollector2(Links.Unsync, next);
1453                 collector.Collect(linksToConnect[i]);
1454                 results = results.And(next);
1455             }
1456             return results.GetSetUInt64Indices();
1457         });
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
1523     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1524     public List<ulong> GetSimilarSequences() => new List<ulong>();
1525
1526     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1527     public void Prediction()
1528     {
1529         //_links
1530         //sequences
1531     }
1532
1533     #region From Triplets
1534
1535     //public static void DeleteSequence(Link sequence)
1536     //{
1537     //}
1538
1539     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1540     public List<ulong> CollectMatchingSequences(ulong[] links)
1541     {
1542         if (links.Length == 1)
1543         {
1544             throw new InvalidOperationException("Подпоследовательности с одним элементом не
1545                 ↪ поддерживаются.");
1546         }
1547         var leftBound = 0;
1548         var rightBound = links.Length - 1;
1549         var left = links[leftBound++];
1550         var right = links[rightBound--];
1551         var results = new List<ulong>();
1552         CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1553         return results;
1554     }
1555
1556     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1557     private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1558         ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1559     {
1560         var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1561         var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1562         if (leftLinkTotalReferers <= rightLinkTotalReferers)
1563         {
1564             var nextLeftLink = middleLinks[leftBound];
1565             var elements = GetRightElements(leftLink, nextLeftLink);
1566             if (leftBound <= rightBound)
1567             {
1568                 for (var i = elements.Length - 1; i >= 0; i--)
1569                 {
1570                     var element = elements[i];
1571                     if (element != 0)
1572                     {
1573                         CollectMatchingSequences(element, leftBound + 1, middleLinks,
1574                             ↪ rightLink, rightBound, ref results);
1575                     }
1576                 }
1577             }
1578             else
1579             {
1580                 for (var i = elements.Length - 1; i >= 0; i--)
1581                 {
1582                     var element = elements[i];
1583                     if (element != 0)
1584                     {
1585                         results.Add(element);
1586                     }
1587                 }
1588             }
1589         }
1590     }
1591
1592     }
1593
1594     }
1595
1596     else

```

```

1587     {
1588         var nextRightLink = middleLinks[rightBound];
1589         var elements = GetLeftElements(rightLink, nextRightLink);
1590         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     });

```

```

1663     }
1664     return true;
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             {
1711                 result[offset + 1] = couple;
1712                 if (++added == 2)
1713                 {
1714                     return false;
1715                 }
1716             }
1717         }
1718         return true;
1719     });
1720     return added > 0;
1721 }
1722
1723 #endregion
1724
1725 #region Walkers
1726
1727 public class PatternMatcher : RightSequenceWalker<ulong>
1728 {
1729     private readonly Sequences _sequences;
1730     private readonly ulong[] _patternSequence;
1731     private readonly HashSet<LinkIndex> _linksInSequence;
1732     private readonly HashSet<LinkIndex> _results;
1733
1734     #region Pattern Match
1735
1736     enum PatternBlockType
1737     {
1738         Undefined,
1739         Gap,
1740         Elements
1741     }

```

```

1742 struct PatternBlock
1743 {
1744     public PatternBlockType Type;
1745     public long Start;
1746     public long Stop;
1747 }
1748
1749 private readonly List<PatternBlock> _pattern;
1750 private int _patternPosition;
1751 private long _sequencePosition;
1752
1753 #endregion
1754
1755 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1756 public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1757     ↳ HashSet<LinkIndex> results)
1758     : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1759 {
1760     _sequences = sequences;
1761     _patternSequence = patternSequence;
1762     _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1763     ↳ _sequences.Constants.Any && x != ZeroOrMany));
1764     _results = results;
1765     _pattern = CreateDetailedPattern();
1766 }
1767
1768 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1769 protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
1770     ↳ base.IsElement(link);
1771
1772 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1773 public bool PatternMatch(LinkIndex sequenceToMatch)
1774 {
1775     _patternPosition = 0;
1776     _sequencePosition = 0;
1777     foreach (var part in Walk(sequenceToMatch))
1778     {
1779         if (!PatternMatchCore(part))
1780         {
1781             break;
1782         }
1783     }
1784     return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1785     ↳ - 1 && _pattern[_patternPosition].Start == 0);
1786 }
1787
1788 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1789 private List<PatternBlock> CreateDetailedPattern()
1790 {
1791     var pattern = new List<PatternBlock>();
1792     var patternBlock = new PatternBlock();
1793     for (var i = 0; i < _patternSequence.Length; i++)
1794     {
1795         if (patternBlock.Type == PatternBlockType.Undefined)
1796         {
1797             if (_patternSequence[i] == _sequences.Constants.Any)
1798             {
1799                 patternBlock.Type = PatternBlockType.Gap;
1800                 patternBlock.Start = 1;
1801                 patternBlock.Stop = 1;
1802             }
1803             else if (_patternSequence[i] == ZeroOrMany)
1804             {
1805                 patternBlock.Type = PatternBlockType.Gap;
1806                 patternBlock.Start = 0;
1807                 patternBlock.Stop = long.MaxValue;
1808             }
1809             else
1810             {
1811                 patternBlock.Type = PatternBlockType.Elements;
1812                 patternBlock.Start = i;
1813                 patternBlock.Stop = i;
1814             }
1815         }
1816         else if (patternBlock.Type == PatternBlockType.Elements)
1817         {
1818             if (_patternSequence[i] == _sequences.Constants.Any)
1819             {
1820                 pattern.Add(patternBlock);
1821                 patternBlock = new PatternBlock();
1822             }
1823         }
1824     }
1825     pattern.Add(patternBlock);
1826 }

```

```

1818         patternBlock = new PatternBlock
1819         {
1820             Type = PatternBlockType.Gap,
1821             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 //    { /* 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 //    if (maximumGap < mininumGap)
1923 //        maximumGap = mininumGap;
1924 //}
1925
1926 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1927 private bool PatternMatchCore(LinkIndex element)
1928 {
1929     if (_patternPosition >= _pattern.Count)
1930     {
1931         _patternPosition = -2;
1932         return false;
1933     }
1934     var currentPatternBlock = _pattern[_patternPosition];
1935     if (currentPatternBlock.Type == PatternBlockType.Gap)
1936     {
1937         //var currentMatchingBlockLength = (_sequencePosition -
1938         ↪ _lastMatchedBlockPosition);
1939         if (_sequencePosition < currentPatternBlock.Start)
1940         {
1941             _sequencePosition++;
1942             return true; // Двигаемся дальше
1943         }
1944         // Это последний блок
1945         if (_pattern.Count == _patternPosition + 1)
1946         {
1947             _patternPosition++;
1948             _sequencePosition = 0;
1949             return false; // Полное соответствие
1950         }
1951         else
1952         {
1953             if (_sequencePosition > currentPatternBlock.Stop)
1954             {
1955                 return false; // Соответствие невозможно
1956             }
1957             var nextPatternBlock = _pattern[_patternPosition + 1];
1958             if (_patternSequence[nextPatternBlock.Start] == element)
1959             {
1960                 if (nextPatternBlock.Start < nextPatternBlock.Stop)
1961                 {
1962                     _patternPosition++;
1963                     _sequencePosition = 1;
1964                 }
1965                 else
1966                 {
1967                     _patternPosition += 2;
1968                     _sequencePosition = 0;
1969                 }
1970             }
1971         }
1972     }
1973     else // currentPatternBlock.Type == PatternBlockType.Elements

```



```

1974         var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1975         if (_patternSequence[patternElementPosition] != element)
1976         {
1977             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     //if (_filterPosition == _patternSequence.Length)
1999     //{
2000         //    _filterPosition = -2; // Длиннее чем нужно
2001         //    return false;
2002     //}
2003     //if (element != _patternSequence[_filterPosition])
2004     //{
2005         //    _filterPosition = -1;
2006         //    return false; // Начинается иначе
2007     //}
2008     //if (_filterPosition == (_patternSequence.Length - 1))
2009     //    return false;
2010     //if (_filterPosition >= 0)
2011     //{
2012         //    if (element == _patternSequence[_filterPosition + 1])
2013         //        _filterPosition++;
2014         //    else
2015         //        return false;
2016     //}
2017     //if (_filterPosition < 0)
2018     //{
2019         //    if (element == _patternSequence[0])
2020         //        _filterPosition = 0;
2021     //}
2022 }
2023
2024 [MethodImpl(MethodImplOptions.AggressiveInlining)]
2025 public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
2026 {
2027     foreach (var sequenceToMatch in sequencesToMatch)
2028     {
2029         if (PatternMatch(sequenceToMatch))
2030         {
2031             _results.Add(sequenceToMatch);
2032         }
2033     }
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 звёздочкой.
37     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
38     /// так же проблема может быть решена при реализации дистанционных триггеров.
39     /// Нужны ли уникальные указатели вообще?
40     /// Что если обращение к информации будет происходить через содержимое всегда?
41     ///
42     /// Писать тесты.
43     ///
44     /// Можно убрать зависимость от конкретной реализации Links,
45     /// на зависимость от абстрактного элемента, который может быть представлен несколькими
46     /// ↪ способами.
47     ///
48     /// Можно ли как-то сделать один общий интерфейс
49     ///
50     /// Блокчейн и/или гит для распределённой записи транзакций.
51     ///
52     /// </remarks>
53     public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
54     ↪ (после завершения реализации Sequences)
55     {
56         /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
57         ↪ связей.</summary>
58         public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
59
60         public SequencesOptions<LinkIndex> Options { get; }
61         public SynchronizedLinks<LinkIndex> Links { get; }
62         private readonly ISynchronization _sync;
63
64         public LinksConstants<LinkIndex> Constants { get; }
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
68         {
69             Links = links;
70             _sync = links.SyncRoot;
71             Options = options;
72             Options.ValidateOptions();
73             Options.InitOptions(Links);
74             Constants = links.Constants;
75         }
76
77         [MethodImpl(MethodImplOptions.AggressiveInlining)]
78         public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
79         ↪ SequencesOptions<LinkIndex>()) { }
80
81         [MethodImpl(MethodImplOptions.AggressiveInlining)]
82         public bool IsSequence(LinkIndex sequence)
83         {
84             return _sync.ExecuteReadOperation(() =>
85             {
86                 if (Options.UseSequenceMarker)

```

```

81         {
82             return Options.MarkedSequenceMatcher.IsMatched(sequence);
83         }
84         return !Links.Unsync.IsPartialPoint(sequence);
85     });
86 }
87
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 private LinkIndex GetSequenceByElements(LinkIndex sequence)
90 {
91     if (Options.UseSequenceMarker)
92     {
93         return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
94     }
95     return sequence;
96 }
97
98 [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) -
166                     ↳ 1;
167             }
168             return Links.Count(any, elementsLink);
169         }
170         return Links.Count(any, restrictions[0]);
171     }
172     throw new NotImplementedException();
173 }
174 #endregion
175 #region Create
176 [MethodImpl(MethodImplOptions.AggressiveInlining)]
177 public LinkIndex Create(ICollection<LinkIndex> restrictions)
178 {
179     return _sync.ExecuteWriteOperation(() =>
180     {
181         if (restrictions.IsNullOrEmpty())
182         {
183             return Constants.Null;
184         }
185         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
186         return CreateCore(restrictions);
187     });
188 }
189 [MethodImpl(MethodImplOptions.AggressiveInlining)]
190 private LinkIndex CreateCore(ICollection<LinkIndex> restrictions)
191 {
192     LinkIndex[] sequence = restrictions.SkipFirst();
193     if (Options.UseIndex)
194     {
195         Options.Index.Add(sequence);
196     }
197     var sequenceRoot = default(LinkIndex);
198     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
199     {
200         var matches = Each(restrictions);
201         if (matches.Count > 0)
202         {
203             sequenceRoot = matches[0];
204         }
205     }
206     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
207     {
208         return CompactCore(sequence);
209     }
210     if (sequenceRoot == default)
211     {
212         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
213     }
214     if (Options.UseSequenceMarker)
215     {
216         return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
217     }
218     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
219 }
220 #endregion
221 #region Each
222 [MethodImpl(MethodImplOptions.AggressiveInlining)]
223 public List<LinkIndex> Each(ICollection<LinkIndex> sequence)
224 {
225     var results = new List<LinkIndex>();
226     var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
227     Each(filler.AddFirstAndReturnConstant, sequence);
228     return results;
229 }
230 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

238 public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<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,
    ↳ Options.SequenceMarkerLink, any));
256                 }
257                 else
258                 {
259                     return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
    ↳ any));
260                 }
261             }
262             if (Options.UseSequenceMarker)
263             {
264                 var sequenceLinkValues = Links.Unsync.GetLink(link);
265                 if (sequenceLinkValues[Constants.SourcePart] ==
    ↳ Options.SequenceMarkerLink)
266                 {
267                     link = sequenceLinkValues[Constants.TargetPart];
268                 }
269             }
270             var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
271             sequence[0] = link;
272             return handler(sequence);
273         }
274         else if (restrictions.Count == 2)
275         {
276             throw new NotImplementedException();
277         }
278         else if (restrictions.Count == 3)
279         {
280             return Links.Unsync.Each(handler, restrictions);
281         }
282         else
283         {
284             var sequence = restrictions.SkipFirst();
285             if (Options.UseIndex && !Options.Index.MightContain(sequence))
286             {
287                 return Constants.Break;
288             }
289             return EachCore(handler, sequence);
290         }
291     });
292 }
293
294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
    ↳ values)
296 {
297     var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
298     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
    ↳ Id.
299     Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
    ↳ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
    ↳ matcher.HandleFullMatched;
300     //if (sequence.Length >= 2)
301     if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
302     {
303         return Constants.Break;
304     }
305     var last = values.Count - 2;
306     for (var i = 1; i < last; i++)
307     {

```

```

308         if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
309             ↪ Constants.Continue)
310         {
311             return Constants.Break;
312         }
313     }
314     if (values.Count >= 3)
315     {
316         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
317             ↪ != Constants.Continue)
318         {
319             return Constants.Break;
320         }
321     }
322     return Constants.Continue;
323 }
324 [MethodImpl(MethodImplOptions.AggressiveInlining)]
325 private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
326 ↪ left, LinkIndex right)
327 {
328     return Links.Unsync.Each(doublet =>
329     {
330         var doubletIndex = doublet[Constants.IndexPart];
331         if (StepRight(handler, doubletIndex, right) != Constants.Continue)
332         {
333             return Constants.Break;
334         }
335         if (left != doubletIndex)
336         {
337             return PartialStepRight(handler, doubletIndex, right);
338         }
339         return Constants.Continue;
340     }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
341 }
342 [MethodImpl(MethodImplOptions.AggressiveInlining)]
343 private LinkIndex StepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
344 ↪ LinkIndex right) => Links.Unsync.Each(rightStep => TryStepRightUp(handler, right,
345 ↪ rightStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, left,
346 ↪ Constants.Any));
347 [MethodImpl(MethodImplOptions.AggressiveInlining)]
348 private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
349 ↪ right, LinkIndex stepFrom)
350 {
351     var upStep = stepFrom;
352     var firstSource = Links.Unsync.GetTarget(upStep);
353     while (firstSource != right && firstSource != upStep)
354     {
355         upStep = firstSource;
356         firstSource = Links.Unsync.GetSource(upStep);
357     }
358     if (firstSource == right)
359     {
360         return handler(new LinkAddress<LinkIndex>(stepFrom));
361     }
362     return Constants.Continue;
363 }
364 [MethodImpl(MethodImplOptions.AggressiveInlining)]
365 private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
366 ↪ LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
367 ↪ leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
368 ↪ right));
369 [MethodImpl(MethodImplOptions.AggressiveInlining)]
370 private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
371 ↪ left, LinkIndex stepFrom)
372 {
373     var upStep = stepFrom;
374     var firstTarget = Links.Unsync.GetSource(upStep);
375     while (firstTarget != left && firstTarget != upStep)
376     {
377         upStep = firstTarget;
378         firstTarget = Links.Unsync.GetTarget(upStep);
379     }
380     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);

```

```

451         }
452         Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
453     }
454     ClearGarbage(sequenceElementsContents.Source);
455     ClearGarbage(sequenceElementsContents.Target);
456 }
457 else
458 {
459     if (Options.UseSequenceMarker)
460     {
461         var sequenceElements = GetSequenceElements(sequence);
462         var sequenceLink = GetSequenceByElements(sequenceElements);
463         var newSequenceElements = GetSequenceElements(newSequence);
464         var newSequenceLink = GetSequenceByElements(newSequenceElements);
465         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
466         {
467             if (sequenceLink != Constants.Null)
468             {
469                 Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
470             }
471             Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
472         }
473     }
474     else
475     {
476         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
477         {
478             Links.Unsync.MergeAndDelete(sequence, newSequence);
479         }
480     }
481 }
482 }
483
484 #endregion
485
486 #region Delete
487
488 [MethodImpl(MethodImplOptions.AggressiveInlining)]
489 public void Delete(ICollection<LinkIndex> restrictions)
490 {
491     _sync.ExecuteWriteOperation(() =>
492     {
493         var sequence = restrictions.SkipFirst();
494         // TODO: Check all options only ones before loop execution
495         foreach (var linkToDelete in Each(sequence))
496         {
497             DeleteOneCore(linkToDelete);
498         }
499     });
500 }
501
502 [MethodImpl(MethodImplOptions.AggressiveInlining)]
503 private void DeleteOneCore(LinkIndex link)
504 {
505     if (Options.UseGarbageCollection)
506     {
507         var sequenceElements = GetSequenceElements(link);
508         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
509         var sequenceLink = GetSequenceByElements(sequenceElements);
510         if (Options.UseCascadeDelete || CountUsages(link) == 0)
511         {
512             if (sequenceLink != Constants.Null)
513             {
514                 Links.Unsync.Delete(sequenceLink);
515             }
516             Links.Unsync.Delete(link);
517         }
518         ClearGarbage(sequenceElementsContents.Source);
519         ClearGarbage(sequenceElementsContents.Target);
520     }
521     else
522     {
523         if (Options.UseSequenceMarker)
524         {
525             var sequenceElements = GetSequenceElements(link);
526             var sequenceLink = GetSequenceByElements(sequenceElements);
527             if (Options.UseCascadeDelete || CountUsages(link) == 0)
528             {

```



```

529         if (sequenceLink != Constants.Null)
530         {
531             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 /// TODO0: 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     }
609 }

```

```

605         ClearGarbage(contents.Source);
606         ClearGarbage(contents.Target);
607     }
608 }
609
610 #endregion
611
612 #region Walkers
613
614 [MethodImpl(MethodImplOptions.AggressiveInlining)]
615 public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
616 {
617     return _sync.ExecuteReadOperation(() =>
618     {
619         var links = Links.Unsync;
620         foreach (var part in Options.Walker.Walk(sequence))
621         {
622             if (!handler(part))
623             {
624                 return false;
625             }
626         }
627         return true;
628     });
629 }
630
631 public class Matcher : RightSequenceWalker<LinkIndex>
632 {
633     private readonly Sequences _sequences;
634     private readonly IList<LinkIndex> _patternSequence;
635     private readonly HashSet<LinkIndex> _linksInSequence;
636     private readonly HashSet<LinkIndex> _results;
637     private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
638     private readonly HashSet<LinkIndex> _readAsElements;
639     private int _filterPosition;
640
641     [MethodImpl(MethodImplOptions.AggressiveInlining)]
642     public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
643         ↪ HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
644         ↪ HashSet<LinkIndex> readAsElements = null)
645         : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
646     {
647         _sequences = sequences;
648         _patternSequence = patternSequence;
649         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
650             ↪ _links.Constants.Any && x != ZeroOrMany));
651         _results = results;
652         _stopableHandler = stopableHandler;
653         _readAsElements = readAsElements;
654     }
655
656     [MethodImpl(MethodImplOptions.AggressiveInlining)]
657     protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
658         ↪ (_readAsElements != null && _readAsElements.Contains(link)) ||
659         ↪ _linksInSequence.Contains(link);
660
661     [MethodImpl(MethodImplOptions.AggressiveInlining)]
662     public bool FullMatch(LinkIndex sequenceToMatch)
663     {
664         _filterPosition = 0;
665         foreach (var part in Walk(sequenceToMatch))
666         {
667             if (!FullMatchCore(part))
668             {
669                 break;
670             }
671         }
672         return _filterPosition == _patternSequence.Count;
673     }
674
675     [MethodImpl(MethodImplOptions.AggressiveInlining)]
676     private bool FullMatchCore(LinkIndex element)
677     {
678         if (_filterPosition == _patternSequence.Count)
679         {
680             _filterPosition = -2; // Длиннее чем нужно
681             return false;
682         }
683         if (_patternSequence[_filterPosition] != _links.Constants.Any
684             && element != _patternSequence[_filterPosition])

```

```

680     {
681         _filterPosition = -1;
682         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     {

```

```

758         if (element == _patternSequence[0])
759         {
760             _filterPosition = 0;
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
812 #endregion
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];

```

```

18         finalSequence[i] = part.Length == 1 ? part[0] :
           ↪ sequences.Create(part.ShiftRight());
19     }
20     return sequences.Create(finalSequence.ShiftRight());
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
           ↪ LinkAddress<TLink>(sequence));
29     return list;
30 }
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
           ↪ ILinks<TLink> must contain GetConstants function.
19     {
20         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪ EqualityComparer<TLink>.Default;
21
22         public TLink SequenceMarkerLink
23         {
24             [MethodImpl(MethodImplOptions.AggressiveInlining)]
25             get;
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             set;
28         }
29
30         public bool UseCascadeUpdate
31         {
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             get;
34             [MethodImpl(MethodImplOptions.AggressiveInlining)]
35             set;
36         }
37
38         public bool UseCascadeDelete
39         {
40             [MethodImpl(MethodImplOptions.AggressiveInlining)]
41             get;
42             [MethodImpl(MethodImplOptions.AggressiveInlining)]
43             set;
44         }
45
46         public bool UseIndex
47         {
48             [MethodImpl(MethodImplOptions.AggressiveInlining)]
49             get;
50             [MethodImpl(MethodImplOptions.AggressiveInlining)]
51             set;
52         } // TODO: Update Index on sequence update/delete.
53
54         public bool UseSequenceMarker
55         {
56             [MethodImpl(MethodImplOptions.AggressiveInlining)]
57             get;
58             [MethodImpl(MethodImplOptions.AggressiveInlining)]
59             set;
60         }

```

```

61
62 public bool UseCompression
63 {
64     [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     }
215 }

```

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 }
```



```

18     private readonly Func<TLink, bool> _isElement;
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
22         ↪ base(links) => _isElement = isElement;
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
26         ↪ _links.IsPartialPoint;
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     public TLink[] ToArray(TLink sequence)
33     {
34         var length = 1;
35         var array = new TLink[length];
36         array[0] = sequence;
37         if (_isElement(sequence))
38         {
39             return array;
40         }
41         bool hasElements;
42         do
43         {
44             length *= 2;
45             #if USEARRAYPOOL
46                 var nextArray = ArrayPool.Allocate<ulong>(length);
47             #else
48                 var nextArray = new TLink[length];
49             #endif
50             hasElements = false;
51             for (var i = 0; i < array.Length; i++)
52             {
53                 var candidate = array[i];
54                 if (_equalityComparer.Equals(array[i], default))
55                 {
56                     continue;
57                 }
58                 var doubletOffset = i * 2;
59                 if (_isElement(candidate))
60                 {
61                     nextArray[doubletOffset] = candidate;
62                 }
63                 else
64                 {
65                     var links = _links;
66                     var link = links.GetLink(candidate);
67                     var linkSource = links.GetSource(link);
68                     var linkTarget = links.GetTarget(link);
69                     nextArray[doubletOffset] = linkSource;
70                     nextArray[doubletOffset + 1] = linkTarget;
71                     if (!hasElements)
72                     {
73                         hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
74                     }
75                 }
76             }
77             #if USEARRAYPOOL
78                 if (array.Length > 1)
79                 {
80                     ArrayPool.Free(array);
81                 }
82             #endif
83             array = nextArray;
84         } while (hasElements);
85         var filledElementsCount = CountFilledElements(array);
86         if (filledElementsCount == array.Length)
87         {
88             return array;
89         }
90         else
91         {
92             return CopyFilledElements(array, filledElementsCount);
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    }
52 }

```

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
85             //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
86             ↳ substitutedHandler, Unsync.Trigger);
87         //}
88     }

```

```
77 }
```

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         }
78     }
79 }
```

```

66     if (links.Exists(linkIndex))
67     {
68         if (visited.Add(linkIndex))
69         {
70             sb.Append('(');
71             var link = new Link<ulong>(links.GetLink(linkIndex));
72             if (renderIndex)
73             {
74                 sb.Append(link.Index);
75                 sb.Append(':');
76             }
77             if (link.Source == link.Index)
78             {
79                 sb.Append(link.Index);
80             }
81             else
82             {
83                 var source = new Link<ulong>(links.GetLink(link.Source));
84                 if (isElement(source))
85                 {
86                     appendElement(sb, source);
87                 }
88                 else
89                 {
90                     links.AppendStructure(sb, visited, source.Index, isElement,
91                                     ↪ appendElement, renderIndex);
92                 }
93             }
94             sb.Append(' ');
95             if (link.Target == link.Index)
96             {
97                 sb.Append(link.Index);
98             }
99             else
100             {
101                 var target = new Link<ulong>(links.GetLink(link.Target));
102                 if (isElement(target))
103                 {
104                     appendElement(sb, target);
105                 }
106                 else
107                 {
108                     links.AppendStructure(sb, visited, target.Index, isElement,
109                                     ↪ appendElement, renderIndex);
110                 }
111             }
112             sb.Append(')');
113         }
114         else
115         {
116             if (renderDebug)
117             {
118                 sb.Append('*');
119             }
120             sb.Append(linkIndex);
121         }
122     }
123     else
124     {
125         if (renderDebug)
126         {
127             sb.Append('~');
128         }
129         sb.Append(linkIndex);
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         /// Или
42         ///
43         /// public struct TransitionHeader
44         /// {
45         ///     public ulong TransactionIdCombined;
46         ///     public ulong TimestampCombined;
47         ///
48         ///     public ulong TransactionId
49         ///     {
50         ///         get
51         ///         {
52         ///             return (ulong) mask & TransactionIdCombined;
53         ///         }
54         ///     }
55         ///
56         ///     public UniqueTimestamp Timestamp
57         ///     {
58         ///         get
59         ///         {
60         ///             return (UniqueTimestamp)mask & TransactionIdCombined;
61         ///         }
62         ///     }
63         ///
64         ///     public TransactionItemType Type
65         ///     {
66         ///         get
67         ///         {
68         ///             // Использовать по одному биту из TransactionId и Timestamp,
69         ///             // для значения в 2 бита, которое представляет тип операции
70         ///             throw new NotImplementedException();
71         ///         }
72         ///     }
73         /// }
74         ///
75         /// private struct Transition
76         /// {
77         ///     public TransitionHeader Header;
78         ///     public Link Source;
79         ///     public Link Linker;
80         ///     public Link Target;
81         /// }
82         ///
83         /// </remarks>
84         public struct Transition : IEquatable<Transition>
85         {
86             public static readonly long Size = Structure<Transition>.Size;
87

```



```

88
89 public readonly ulong TransactionId;
90 public readonly Link<ulong> Before;
91 public readonly Link<ulong> After;
92 public readonly Timestamp Timestamp;
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
    ↳ 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
    ↳ transactionId, Link<ulong> before) : this(uniqueTimestampFactory, transactionId,
    ↳ before, default) { }
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
    ↳ transactionId) : this(uniqueTimestampFactory, transactionId, default, default) {
    ↳ }
108
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
    ↳ {After}";
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 public override bool Equals(object obj) => obj is Transition transition ?
    ↳ Equals(transition) : false;
114
115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 public override int GetHashCode() => (TransactionId, Before, After,
    ↳ Timestamp).GetHashCode();
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 public bool Equals(Transition other) => TransactionId == other.TransactionId &&
    ↳ Before == other.Before && After == other.After && Timestamp == other.Timestamp;
120
121 [MethodImpl(MethodImplOptions.AggressiveInlining)]
122 public static bool operator ==(Transition left, Transition right) =>
    ↳ left.Equals(right);
123
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 public static bool operator !=(Transition left, Transition right) => !(left ==
    ↳ right);
126 }
127
128 /// <remarks>
129 /// Другие варианты реализации транзакций (атомарности):
130 /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
    ↳ Target)) и индексов.
131 /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
    ↳ потребуется решить вопрос
132 /// со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
    ↳ пересечениями идентификаторов.
133 ///
134 /// Где хранить промежуточный список транзакций?
135 ///
136 /// В оперативной памяти:
137 /// Минусы:
138 /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
139 /// так как нужно отдельно выделять память под список трансформаций.
140 /// 2. Выделенной оперативной памяти может не хватить, в том случае,
141 /// если транзакция использует слишком много трансформаций.
142 /// -> Можно использовать жёсткий диск для слишком длинных транзакций.
143 /// -> Максимальный размер списка трансформаций можно ограничить / задать
    ↳ константой.
144 /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
    ↳ создавая задержку.
145 ///
146 /// На жёстком диске:
147 /// Минусы:
148 /// 1. Длительный отклик, на запись каждой трансформации.
149 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.

```

```

150     /// -> Это может решаться упаковкой/исключением дублирующих операций.
151     /// -> Также это может решаться тем, что короткие транзакции вообще
152     /// не будут записываться в случае отката.
153     /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
    ↪ 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,
    ↪ Transaction transaction)
206         {
207             layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
208             layer._currentTransactionTransitions = transaction._transitions;
209             layer._currentTransaction = transaction;
210         }
211
212         [MethodImpl(MethodImplOptions.AggressiveInlining)]
213         public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
214         {
215             if (transaction.IsReverted)
216             {
217                 throw new InvalidOperationException("Transation is reverted.");
218             }
219             if (transaction.IsCommitted)
220             {
221                 throw new InvalidOperationException("Transation is committed.");
222             }
223         }
224
225         [MethodImpl(MethodImplOptions.AggressiveInlining)]
226         protected override void Dispose(bool manual, bool wasDisposed)

```

```

227     {
228         if (!wasDisposed && _layer != null && !_layer.Disposable.IsDisposed)
229         {
230             if (!IsCommitted && !IsReverted)
231             {
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];

```

```

303     var beforeLink = new Link<ulong>(_links.GetLink(linkIndex));
304     linkIndex = _links.Update(restrictions, substitution);
305     var afterLink = new Link<ulong>(_links.GetLink(linkIndex));
306     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
307         ↪ beforeLink, afterLink));
308     return linkIndex;
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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
320 private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
321     ↪ _transitions;
322 [MethodImpl(MethodImplOptions.AggressiveInlining)]
323 private void CommitTransition(Transition transition)
324 {
325     if (_currentTransaction != null)
326     {
327         Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
328     }
329     var transitions = GetCurrentTransitions();
330     transitions.Enqueue(transition);
331 }
332 [MethodImpl(MethodImplOptions.AggressiveInlining)]
333 private void RevertTransition(Transition transition)
334 {
335     if (transition.After.IsNull()) // Revert Deletion with Creation
336     {
337         _links.Create();
338     }
339     else if (transition.Before.IsNull()) // Revert Creation with Deletion
340     {
341         _links.Delete(transition.After.Index);
342     }
343     else // Revert Update
344     {
345         _links.Update(new[] { transition.After.Index, transition.Before.Source,
346             ↪ transition.Before.Target });
347     }
348 }
349 [MethodImpl(MethodImplOptions.AggressiveInlining)]
350 private void ResetCurrentTransation()
351 {
352     _currentTransactionId = 0;
353     _currentTransactionTransitions = null;
354     _currentTransaction = null;
355 }
356 [MethodImpl(MethodImplOptions.AggressiveInlining)]
357 private void PushTransitions()
358 {
359     if (_log == null || _transitions == null)
360     {
361         return;
362     }
363     for (var i = 0; i < _transitions.Count; i++)
364     {
365         var transition = _transitions.Dequeue();
366         _log.Write(transition);
367         _lastCommittedTransition = transition;
368     }
369 }
370 [MethodImpl(MethodImplOptions.AggressiveInlining)]
371 private void TransitionsPusher()
372 {
373 }

```

```

377         while (!Disposable.IsDisposed && _transitionsPusher != null)
378         {
379             Thread.Sleep(DefaultPushDelay);
380             PushTransitions();
381         }
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         }
31     }
32 }

```

```

26         return _links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
27     }
28 }
29 }

```

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 }

```

```

31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 public void Init()
33 {
34     if (!_initialized)
35     {
36         return;
37     }
38     _initialized = true;
39     var firstLink = _links.CreatePoint();
40     if (firstLink != FirstCharLink)
41     {
42         _links.Delete(firstLink);
43     }
44     else
45     {
46         for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
47         {
48             // From NIL to It (NIL -> Character) transformation meaning, (or infinite
49             // ↪ amount of NIL characters before actual Character)
50             var createdLink = _links.CreatePoint();
51             _links.Update(createdLink, firstLink, createdLink);
52             if (createdLink != i)
53             {
54                 throw new InvalidOperationException("Unable to initialize UTF 16
55                 ↪ table.");
56             }
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)]

```

```

105 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
106 {
107     // char array to ulong array
108     var linksSequence = new ulong[count];
109     for (var i = 0; i < count; i++)
110     {
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)

```



```

181         {
182             relativeLength++;
183             absoluteLength++;
184         }
185     }
186     // 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))

```

```

30         {
31             throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
↳ not a unicode sequence.");
32         }
33         var sequence = _links.GetSource(source);
34         var charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.
↳ Convert).ToArray();
35         return new string(charArray);
36     }
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>,
↳ ICriterionMatcher<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
↳ EqualityComparer<TLink>.Default;
12
13         private readonly TLink _unicodeSymbolMarker;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public UnicodeSymbolCriterionMatcher(ILinks<TLink> links, TLink unicodeSymbolMarker) :
↳ base(links) => _unicodeSymbolMarker = unicodeSymbolMarker;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public bool IsMatched(TLink link) => _equalityComparer.Equals(_links.GetTarget(link),
↳ _unicodeSymbolMarker);
20     }
21 }

```

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>,
↳ IConverter<TLink, char>
11     {
12         private static readonly UncheckedConverter<TLink, char> _addressToCharConverter =
↳ UncheckedConverter<TLink, char>.Default;
13
14         private readonly IConverter<TLink> _numberToAddressConverter;
15         private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
↳ numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
↳ base(links)
19         {
20             _numberToAddressConverter = numberToAddressConverter;
21             _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public char Convert(TLink source)
26         {
27             if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
28             {
29                 throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
↳ not a unicode symbol.");
30             }
31             return _addressToCharConverter.Convert(_numberToAddressConverter.Convert(_links.GetS
↳ ource(source)));
32         }
33     }
34 }

```

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

```

```

90     {
91         for (int i = 0; i < N; i++)
92         {
93             result = comparer.Compare(x, y) == 0;
94         }
95     });
96
97     Assert.True(ts2 < ts1);
98     Assert.True(ts3 < ts2);
99     Assert.True(ts5 < ts4);
100    Assert.True(ts5 < ts6);
101
102    Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {result}");
103 }
104 }
105 }

```

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

```

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

```

Enim sit amet venenatis urna cursus eget nunc scelerisque viverra.
Amet venenatis urna cursus eget nunc scelerisque viverra mauris in.
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);
```

```

127     var sequenceToItsLocalElementLevelsConverter = new
        ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
128     ↳ linkToItsFrequencyNumberConverter);
129     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
        ↳ sequenceToItsLocalElementLevelsConverter);
130
131     var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
        ↳ Walker = new LeveledSequenceWalker<ulong>(links) });
132
133     ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
        ↳ index, optimalVariantConverter);
134 }
135
136 private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
        ↳ SequenceToItsLocalElementLevelsConverter<ulong>
137     ↳ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
138     ↳ OptimalVariantConverter<ulong> optimalVariantConverter)
139 {
140     index.Add(sequence);
141
142     var optimalVariant = optimalVariantConverter.Convert(sequence);
143
144     var readSequence1 = sequences.ToList(optimalVariant);
145
146     Assert.True(sequence.SequenceEqual(readSequence1));
147 }
148
149 [Fact]
150 public static void SavedSequencesOptimizationTest()
151 {
152     LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
        ↳ (long.MaxValue + 1UL, ulong.MaxValue));
153
154     using (var memory = new HeapResizableDirectMemory())
155     using (var disposableLinks = new UInt64UnitedMemoryLinks(memory,
        ↳ UInt64UnitedMemoryLinks.DefaultLinksSizeStep, constants, useAvlBasedIndex:
        ↳ false))
156     {
157         var links = new UInt64Links(disposableLinks);
158
159         var root = links.CreatePoint();
160
161         //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
162         var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
163
164         var unicodeSymbolMarker = links.GetOrCreate(root,
        ↳ addressToNumberConverter.Convert(1));
165         var unicodeSequenceMarker = links.GetOrCreate(root,
        ↳ addressToNumberConverter.Convert(2));
166
167         var totalSequenceSymbolFrequencyCounter = new
        ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links);
168         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
        ↳ totalSequenceSymbolFrequencyCounter);
169         var index = new
        ↳ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
170         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
171         var sequenceToItsLocalElementLevelsConverter = new
        ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
        ↳ linkToItsFrequencyNumberConverter);
172         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
        ↳ sequenceToItsLocalElementLevelsConverter);
173
174         var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
        ↳ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
175
176         var unicodeSequencesOptions = new SequencesOptions<ulong>()
177         {
178             UseSequenceMarker = true,
179             SequenceMarkerLink = unicodeSequenceMarker,
180             UseIndex = true,
181             Index = index,
182             LinksToSequenceConverter = optimalVariantConverter,
183             Walker = walker,
184             UseGarbageCollection = true
185         };

```



```

184
185     var unicodeSequences = new Sequences.Sequences(new
    ↪ 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> {
    ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
24
25                 var sequence = new ulong[sequenceLength];
26                 for (var i = 0; i < sequenceLength; i++)
27                 {
28                     sequence[i] = links.Create();
29                 }
30
31                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
32
33                 var sw1 = Stopwatch.StartNew();
34                 var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
35
36                 var sw2 = Stopwatch.StartNew();
37                 var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();
38
39                 var sw3 = Stopwatch.StartNew();
40                 var readSequence2 = new List<ulong>();
41                 SequenceWalker.WalkRight(balancedVariant,
42                                         links.GetSource,
43                                         links.GetTarget,
44                                         links.IsPartialPoint,
45                                         readSequence2.Add);
46                 sw3.Stop();

```

```

47
48     Assert.True(sequence.SequenceEqual(readSequence1));
49
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.Memory.United.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 UInt64UnitedMemoryLinks(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(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
29                 using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
30                     ↳ UInt64UnitedMemoryLinks.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(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
46                     using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
47                         ↳ UInt64UnitedMemoryLinks.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                     }
58                 }
59             }
60         }
61     }
62 }

```

```

55     memoryAdapter.Each(foundLink =>
56     {
57         resultLink = foundLink[_constants.IndexPart];
58         return _constants.Break;
59     }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
60     Assert.True(resultLink == link);
61     Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
62     memoryAdapter.Delete(link);
63 }
64 }
65 }

```

1.120 ./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.Memory.United.Generic;
7  using Platform.Data.Doublets.Memory.United.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<UInt64UnitedMemoryLinks>();
31                 var instance = scope.Use<ILinks<ulong>>();
32                 Assert.IsType<UInt64UnitedMemoryLinks>(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                 ↪ UnitedMemoryLinks<ulong>>>())
51             {
52                 var links = scope.Use<ILinks<ulong>>();
53                 Assert.IsType<UnitedMemoryLinks<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             //[Fact]
64             //public void CUDTest()
65             //{
66             //    var tempFilename = Path.GetTempFileName();
67
68             //    const long sequenceLength = 8;
69
70             //    const ulong itself = LinksConstants.Itself;
71
72             //    using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
73             //        ↳ DefaultLinksSizeStep))
74             //    {
75             //        using (var links = new Links(memoryAdapter))
76             //        {
77             //            var sequence = new ulong[sequenceLength];
78             //            for (var i = 0; i < sequenceLength; i++)
79             //            {
80             //                sequence[i] = links.Create(itself, itself);
81
82             //                SequencesOptions o = new SequencesOptions();
83
84             //                // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
85             //                o.

```

```

86 //         var sw1 = Stopwatch.StartNew();
87 //         var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
88
89 //         var sw2 = Stopwatch.StartNew();
90 //         var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
91
92 //         Assert.True(results1.Count > results2.Length);
93 //         Assert.True(sw1.Elapsed > sw2.Elapsed);
94
95 //         for (var i = 0; i < sequenceLength; i++)
96 //             links.Delete(sequence[i]);
97 //     }
98
99 //     File.Delete(tempFilename);
100 // }
101
102 [Fact]
103 public static void AllVariantsSearchTest()
104 {
105     const long sequenceLength = 8;
106
107     using (var scope = new TempLinksTestScope(useSequences: true))
108     {
109         var links = scope.Links;
110         var sequences = scope.Sequences;
111
112         var sequence = new ulong[sequenceLength];
113         for (var i = 0; i < sequenceLength; i++)
114         {
115             sequence[i] = links.Create();
116         }
117
118         var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
119
120         //for (int i = 0; i < createResults.Length; i++)
121         //    sequences.Create(createResults[i]);
122
123         var sw0 = Stopwatch.StartNew();
124         var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
125
126         var sw1 = Stopwatch.StartNew();
127         var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
128
129         var sw2 = Stopwatch.StartNew();
130         var searchResults2 = sequences.Each1(sequence); sw2.Stop();
131
132         var sw3 = Stopwatch.StartNew();
133         var searchResults3 = sequences.Each(sequence.ShiftRight()); sw3.Stop();
134
135         var intersection0 = createResults.Intersect(searchResults0).ToList();
136         Assert.True(intersection0.Count == searchResults0.Count);
137         Assert.True(intersection0.Count == createResults.Length);
138
139         var intersection1 = createResults.Intersect(searchResults1).ToList();
140         Assert.True(intersection1.Count == searchResults1.Count);
141         Assert.True(intersection1.Count == createResults.Length);
142
143         var intersection2 = createResults.Intersect(searchResults2).ToList();
144         Assert.True(intersection2.Count == searchResults2.Count);
145         Assert.True(intersection2.Count == createResults.Length);
146
147         var intersection3 = createResults.Intersect(searchResults3).ToList();
148         Assert.True(intersection3.Count == searchResults3.Count);
149         Assert.True(intersection3.Count == createResults.Length);
150
151         for (var i = 0; i < sequenceLength; i++)
152         {
153             links.Delete(sequence[i]);
154         }
155     }
156 }
157
158 [Fact]
159 public static void BalancedVariantSearchTest()
160 {
161     const long sequenceLength = 200;
162
163     using (var scope = new TempLinksTestScope(useSequences: true))
164     {
165         var links = scope.Links;

```

```

166     var sequences = scope.Sequences;
167
168     var sequence = new ulong[sequenceLength];
169     for (var i = 0; i < sequenceLength; i++)
170     {
171         sequence[i] = links.Create();
172     }
173
174     var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
175
176     var sw1 = Stopwatch.StartNew();
177     var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
178
179     var sw2 = Stopwatch.StartNew();
180     var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
181
182     var sw3 = Stopwatch.StartNew();
183     var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
184
185     // На количестве в 200 элементов это будет занимать вечность
186     //var sw4 = Stopwatch.StartNew();
187     //var searchResults4 = sequences.Each(sequence); sw4.Stop();
188
189     Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
190
191     Assert.True(searchResults3.Count == 1 && balancedVariant ==
192         ↪ searchResults3.First());
193
194     //Assert.True(sw1.Elapsed < sw2.Elapsed);
195
196     for (var i = 0; i < sequenceLength; i++)
197     {
198         links.Delete(sequence[i]);
199     }
200 }
201
202 [Fact]
203 public static void AllPartialVariantsSearchTest()
204 {
205     const long sequenceLength = 8;
206
207     using (var scope = new TempLinksTestScope(useSequences: true))
208     {
209         var links = scope.Links;
210         var sequences = scope.Sequences;
211
212         var sequence = new ulong[sequenceLength];
213         for (var i = 0; i < sequenceLength; i++)
214         {
215             sequence[i] = links.Create();
216         }
217
218         var createResults = sequences.CreateAllVariants2(sequence);
219
220         //var createResultsStrings = createResults.Select(x => x + ": " +
221         ↪ sequences.FormatSequence(x)).ToList();
222         //Global.Trash = createResultsStrings;
223
224         var partialSequence = new ulong[sequenceLength - 2];
225
226         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
227
228         var sw1 = Stopwatch.StartNew();
229         var searchResults1 =
230         ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
231
232         var sw2 = Stopwatch.StartNew();
233         var searchResults2 =
234         ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
235
236         //var sw3 = Stopwatch.StartNew();
237         //var searchResults3 =
238         ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
239
240         var sw4 = Stopwatch.StartNew();
241         var searchResults4 =
242         ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();

```

```

239 //Global.Trash = searchResults3;
240
241 //var searchResults1Strings = searchResults1.Select(x => x + ": " +
    ↳ 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[]

```

```

315     {
316         e1, e2, e1, e2 // mama / papa
317     };
318
319     var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
320
321     var balancedVariant = balancedVariantConverter.Convert(sequence);
322
323     // 1: [1]
324     // 2: [2]
325     // 3: [1,2]
326     // 4: [1,2,1,2]
327
328     var doublet = links.GetSource(balancedVariant);
329
330     var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
331
332     Assert.True(matchedSequences1.Count == 0);
333
334     var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
335
336     Assert.True(matchedSequences2.Count == 0);
337
338     var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
339
340     Assert.True(matchedSequences3.Count == 0);
341
342     var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
343
344     Assert.Contains(doublet, matchedSequences4);
345     Assert.Contains(balancedVariant, matchedSequences4);
346
347     for (var i = 0; i < sequence.Length; i++)
348     {
349         links.Delete(sequence[i]);
350     }
351 }
352 }
353
354 [Fact]
355 public static void IndexTest()
356 {
357     using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
358 ↪ true }, useSequences: true))
359     {
360         var links = scope.Links;
361         var sequences = scope.Sequences;
362         var index = sequences.Options.Index;
363
364         var e1 = links.Create();
365         var e2 = links.Create();
366
367         var sequence = new[]
368         {
369             e1, e2, e1, e2 // mama / papa
370         };
371
372         Assert.False(index.MightContain(sequence));
373
374         index.Add(sequence);
375
376         Assert.True(index.MightContain(sequence));
377     }
378 }
379
380 /// <summary>Imported from https://raw.githubusercontent.com/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?

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

386

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

388

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

390

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

392

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

394

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

```

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-animat
    ↳  ion-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 Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
467 Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
468 Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
469 Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
470
471
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);

```

```

526 var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
527     ↳ totalSequenceSymbolFrequencyCounter);
528 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache3);
529
530 var sequenceToItsLocalElementLevelsConverter = new
531     ↳ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
532     ↳ linkToItsFrequencyNumberConverter);
533 var optimalVariantConverter = new
534     ↳ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
535     ↳ sequenceToItsLocalElementLevelsConverter);
536
537 var compressed1 = new ulong[arrays.Length];
538 var compressed2 = new ulong[arrays.Length];
539 var compressed3 = new ulong[arrays.Length];
540
541 var START = 0;
542 var END = arrays.Length;
543
544 //for (int i = START; i < END; i++)
545 //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
546
547 var initialCount1 = scope2.Links.Unsync.Count();
548
549 var sw1 = Stopwatch.StartNew();
550
551 for (int i = START; i < END; i++)
552 {
553     linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
554     compressed1[i] = compressor1.Convert(arrays[i]);
555 }
556
557 var elapsed1 = sw1.Elapsed;
558
559 var balancedVariantConverter2 = new
560     ↳ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
561
562 var initialCount2 = scope2.Links.Unsync.Count();
563
564 var sw2 = Stopwatch.StartNew();
565
566 for (int i = START; i < END; i++)
567 {
568     compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
569 }
570
571 var elapsed2 = sw2.Elapsed;
572
573 for (int i = START; i < END; i++)
574 {
575     linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
576 }
577
578 var initialCount3 = scope3.Links.Unsync.Count();
579
580 var sw3 = Stopwatch.StartNew();
581
582 for (int i = START; i < END; i++)
583 {
584     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
585     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
586 }
587
588 var elapsed3 = sw3.Elapsed;
589
590 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
591     ↳ Optimal variant: {elapsed3}");
592
593 // Assert.True(elapsed1 > elapsed2);
594
595 // Checks
596 for (int i = START; i < END; i++)
597 {
598     var sequence1 = compressed1[i];
599     var sequence2 = compressed2[i];
600     var sequence3 = compressed3[i];
601
602     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
603         ↳ scope1.Links.Unsync);

```

```

597         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
598             ↪ scope2.Links.Unsync);
599
600         var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
601             ↪ scope3.Links.Unsync);
602
603         var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
604             ↪ link.IsPartialPoint());
605         var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
606             ↪ link.IsPartialPoint());
607         var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
608             ↪ link.IsPartialPoint());
609
610         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
611             ↪ arrays[i].Length > 3)
612             // Assert.False(structure1 == structure2);
613         //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
614             ↪ arrays[i].Length > 3)
615             // Assert.False(structure3 == structure2);
616
617         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
618         Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
619     }
620
621     Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
622         ↪ totalCharacters);
623     Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
624         ↪ totalCharacters);
625     Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
626         ↪ totalCharacters);
627
628     Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
629         ↪ totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
630         ↪ totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
631         ↪ totalCharacters}");
632
633     Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
634         ↪ scope2.Links.Unsync.Count() - initialCount2);
635     Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
636         ↪ scope2.Links.Unsync.Count() - initialCount2);
637
638     var duplicateProvider1 = new
639         ↪ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
640     var duplicateProvider2 = new
641         ↪ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
642     var duplicateProvider3 = new
643         ↪ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
644
645     var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
646     var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
647     var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
648
649     var duplicates1 = duplicateCounter1.Count();
650
651     ConsoleHelpers.Debug("-----");
652
653     var duplicates2 = duplicateCounter2.Count();
654
655     ConsoleHelpers.Debug("-----");
656
657     var duplicates3 = duplicateCounter3.Count();
658
659     Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
660
661     linkFrequenciesCache1.ValidateFrequencies();
662     linkFrequenciesCache3.ValidateFrequencies();
663 }
664
665 [Fact]
666 public static void CompressionStabilityTest()
667 {
668     // TODO: Fix bug (do a separate test)
669     //const ulong minNumbers = 0;
670     //const ulong maxNumbers = 1000;
671
672     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;

```

```

735 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
736 ↪ {elapsed2}");
737
738 Assert.True(elapsed1 > elapsed2);
739
740 // Checks
741 for (int i = START; i < END; i++)
742 {
743     var sequence1 = compressed1[i];
744     var sequence2 = compressed2[i];
745
746     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
747     {
748         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
749 ↪ scope1.Links);
750
751         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
752 ↪ scope2.Links);
753
754         //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
755 ↪ link.IsPartialPoint());
756         //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
757 ↪ link.IsPartialPoint());
758
759         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
760 ↪ arrays[i].Length > 3)
761         //    Assert.False(structure1 == structure2);
762
763         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
764     }
765 }
766
767 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
768 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
769
770 Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
771 ↪ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
772 ↪ totalCharacters}}");
773
774 Assert.True(scope1.Links.Count() <= scope2.Links.Count());
775
776 //compressor1.ValidateFrequencies();
777 }
778
779 [Fact]
780 public static void RandomNumbersCompressionQualityTest()
781 {
782     const ulong N = 500;
783
784     //const ulong minNumbers = 10000;
785     //const ulong maxNumbers = 20000;
786
787     //var strings = new List<string>();
788
789     //for (ulong i = 0; i < N; i++)
790     //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
791 ↪ maxNumbers).ToString());
792
793     var strings = new List<string>();
794
795     for (ulong i = 0; i < N; i++)
796     {
797         strings.Add(RandomHelpers.Default.NextUInt64().ToString());
798     }
799
800     strings = strings.Distinct().ToList();
801
802     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
803     var totalCharacters = arrays.Select(x => x.Length).Sum();
804
805     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
806 ↪ SequencesOptions<ulong> { UseCompression = true,
807 ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
808     using (var scope2 = new TempLinksTestScope(useSequences: true))
809     {
810         scope1.Links.UseUnicode();

```

```

802     scope2.Links.UseUnicode();
803
804     var compressor1 = scope1.Sequences;
805     var compressor2 = scope2.Sequences;
806
807     var compressed1 = new ulong[arrays.Length];
808     var compressed2 = new ulong[arrays.Length];
809
810     var sw1 = Stopwatch.StartNew();
811
812     var START = 0;
813     var END = arrays.Length;
814
815     for (int i = START; i < END; i++)
816     {
817         compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
818     }
819
820     var elapsed1 = sw1.Elapsed;
821
822     var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
823
824     var sw2 = Stopwatch.StartNew();
825
826     for (int i = START; i < END; i++)
827     {
828         compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
829     }
830
831     var elapsed2 = sw2.Elapsed;
832
833     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
834         ↪ {elapsed2}");
835
836     Assert.True(elapsed1 > elapsed2);
837
838     // Checks
839     for (int i = START; i < END; i++)
840     {
841         var sequence1 = compressed1[i];
842         var sequence2 = compressed2[i];
843
844         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
845         {
846             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
847                 ↪ scope1.Links);
848
849             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
850                 ↪ scope2.Links);
851
852             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
853         }
854     }
855
856     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
857     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
858
859     Debug.WriteLine($"{{(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;

```



```

877     var sequences = scope.Sequences;
878
879     var sequence = new ulong[sequenceLength];
880     for (var i = 0; i < sequenceLength; i++)
881     {
882         sequence[i] = links.Create();
883     }
884
885     var createResults = sequences.CreateAllVariants2(sequence);
886
887     Global.Trash = createResults;
888
889     for (var i = 0; i < sequenceLength; i++)
890     {
891         links.Delete(sequence[i]);
892     }
893 }
894
895 [Fact]
896 public static void AllPossibleConnectionsTest()
897 {
898     const long sequenceLength = 5;
899
900     using (var scope = new TempLinksTestScope(useSequences: true))
901     {
902         var links = scope.Links;
903         var sequences = scope.Sequences;
904
905         var sequence = new ulong[sequenceLength];
906         for (var i = 0; i < sequenceLength; i++)
907         {
908             sequence[i] = links.Create();
909         }
910
911         var createResults = sequences.CreateAllVariants2(sequence);
912         var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
913
914         for (var i = 0; i < 1; i++)
915         {
916             var sw1 = Stopwatch.StartNew();
917             var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
918
919             var sw2 = Stopwatch.StartNew();
920             var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
921
922             var sw3 = Stopwatch.StartNew();
923             var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
924
925             var sw4 = Stopwatch.StartNew();
926             var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
927
928             Global.Trash = searchResults3;
929             Global.Trash = searchResults4; //-V3008
930
931             var intersection1 = createResults.Intersect(searchResults1).ToList();
932             Assert.True(intersection1.Count == createResults.Length);
933
934             var intersection2 = reverseResults.Intersect(searchResults1).ToList();
935             Assert.True(intersection2.Count == reverseResults.Length);
936
937             var intersection0 = searchResults1.Intersect(searchResults2).ToList();
938             Assert.True(intersection0.Count == searchResults2.Count);
939
940             var intersection3 = searchResults2.Intersect(searchResults3).ToList();
941             Assert.True(intersection3.Count == searchResults3.Count);
942
943             var intersection4 = searchResults3.Intersect(searchResults4).ToList();
944             Assert.True(intersection4.Count == searchResults4.Count);
945         }
946
947         for (var i = 0; i < sequenceLength; i++)
948         {
949             links.Delete(sequence[i]);
950         }
951     }
952 }
953
954 [Fact(Skip = "Correct implementation is pending")]
955 public static void CalculateAllUsagesTest()
956

```

```

957 {
958     const long sequenceLength = 3;
959
960     using (var scope = new TempLinksTestScope(useSequences: true))
961     {
962         var links = scope.Links;
963         var sequences = scope.Sequences;
964
965         var sequence = new ulong[sequenceLength];
966         for (var i = 0; i < sequenceLength; i++)
967         {
968             sequence[i] = links.Create();
969         }
970
971         var createResults = sequences.CreateAllVariants2(sequence);
972
973         //var reverseResults =
974         ↪ sequences.CreateAllVariants2(sequence.Reverse().ToArray());
975
976         for (var i = 0; i < 1; i++)
977         {
978             var linksTotalUsages1 = new ulong[links.Count() + 1];
979
980             sequences.CalculateAllUsages(linksTotalUsages1);
981
982             var linksTotalUsages2 = new ulong[links.Count() + 1];
983
984             sequences.CalculateAllUsages2(linksTotalUsages2);
985
986             var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
987             Assert.True(intersection1.Count == linksTotalUsages2.Length);
988         }
989
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]
20         public static void RawNumbersCRUDTest()
21         {
22             UsingWithExternalReferences<byte>(links => links.TestRawNumbersCRUDOperations());
23             UsingWithExternalReferences<ushort>(links => links.TestRawNumbersCRUDOperations());
24             UsingWithExternalReferences<uint>(links => links.TestRawNumbersCRUDOperations());
25             UsingWithExternalReferences<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));

```

```

34         Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
        ↪ tMultipleRandomCreationsAndDeletions(100));
35     }
36
37     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     private static void UsingWithExternalReferences<TLink>(Action<ILinks<TLink>> action)
48     {
49         var constants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
50         using (var dataMemory = new HeapResizableDirectMemory())
51         using (var indexMemory = new HeapResizableDirectMemory())
52         using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory,
        ↪ SplitMemoryLinks<TLink>.DefaultLinksSizeStep, constants))
53         {
54             action(memory);
55         }
56     }
57 }
58 }

```

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.Memory.United.Specific;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public class TempLinksTestScope : DisposableBase
10     {
11         public ILinks<ulong> MemoryAdapter { get; }
12         public SynchronizedLinks<ulong> Links { get; }
13         public Sequences.Sequences Sequences { get; }
14         public string TempFilename { get; }
15         public string TempTransactionLogFilename { get; }
16         private readonly bool _deleteFiles;
17
18         public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
        ↪ useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
        ↪ useLog) { }
19
20         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
        ↪ true, bool useSequences = false, bool useLog = false)
21         {
22             _deleteFiles = deleteFiles;
23             TempFilename = Path.GetTempFileName();
24             TempTransactionLogFilename = Path.GetTempFileName();
25             var coreMemoryAdapter = new UInt64UnitedMemoryLinks(TempFilename);
26             MemoryAdapter = useLog ? (ILinks<ulong>)new
        ↪ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
        ↪ coreMemoryAdapter;
27             Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
28             if (useSequences)
29             {
30                 Sequences = new Sequences.Sequences(Links, sequencesOptions);
31             }
32         }
33
34         protected override void Dispose(bool manual, bool wasDisposed)
35         {
36             if (!wasDisposed)
37             {
38                 Links.Unsync.DisposeIfPossible();
39                 if (_deleteFiles)
40                 {
41                     DeleteFiles();
42                 }
43             }
44         }
45     }

```

```

46     public void DeleteFiles()
47     {
48         File.Delete(TempFilename);
49         File.Delete(TempTransactionLogFilename);
50     }
51 }
52 }

```

1.124 ./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     Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
154 }
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.Memory.United.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         #region Concept
32
33         [Fact]
34         public static void MultipleCreateAndDeleteTest()
35         {
36             using (var scope = new Scope<Types<HeapResizableDirectMemory,
37                 ↳ UInt64UnitedMemoryLinks>>())
38             {
39                 new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeleti
40                 ↳ ons(100);
41             }
42
43         [Fact]
44         public static void CascadeUpdateTest()
45         {
46             var itself = _constants.Itself;
47             using (var scope = new TempLinksTestScope(useLog: true))
48             {
49                 var links = scope.Links;
50
51                 var l1 = links.Create();
52                 var l2 = links.Create();
53
54                 l2 = links.Update(l2, l2, l1, l2);
55
56                 links.CreateAndUpdate(l2, itself);
57                 links.CreateAndUpdate(l2, itself);
58
59                 l2 = links.Update(l2, l1);
60
61                 links.Delete(l2);
62
63                 Global.Trash = links.Count();
64
65                 links.Unsync.DisposeIfPossible(); // Close links to access log
66
67                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
68                 ↳ e.TempTransactionLogFilename);
69             }
70
71         [Fact]
72         public static void BasicTransactionLogTest()
73         {
74             using (var scope = new TempLinksTestScope(useLog: true))
75             {
76                 var links = scope.Links;
77                 var l1 = links.Create();
78                 var l2 = links.Create();
79
80                 Global.Trash = links.Update(l2, l2, l1, l2);
81
82                 links.Delete(l1);
83
84                 links.Unsync.DisposeIfPossible(); // Close links to access log
85
86                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
87                 ↳ e.TempTransactionLogFilename);
88             }
89
90         [Fact]
91         public static void TransactionAutoRevertedTest()
92         {
93             // Auto Reverted (Because no commit at transaction)
94             using (var scope = new TempLinksTestScope(useLog: true))
95             {
96                 var links = scope.Links;
97                 var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
98                 using (var transaction = transactionsLayer.BeginTransaction())

```

```

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
110             ↪ cope.TempTransactionLogFilename);
111         Assert.Single(transitions);
112     }
113
114     [Fact]
115     public static void TransactionUserCodeErrorNoDataSavedTest()
116     {
117         // User Code Error (Autoreverted), no data saved
118         var itself = _constants.Itself;
119
120         TempLinksTestScope lastScope = null;
121         try
122         {
123             using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
124                 ↪ useLog: true))
125             {
126                 var links = scope.Links;
127                 var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
128                     ↪ atorBase<ulong>)links.Unsync).Links;
129                 using (var transaction = transactionsLayer.BeginTransaction())
130                 {
131                     var l1 = links.CreateAndUpdate(itself, itself);
132                     var l2 = links.CreateAndUpdate(itself, itself);
133
134                     l2 = links.Update(l2, l2, l1, l2);
135
136                     links.CreateAndUpdate(l2, itself);
137                     links.CreateAndUpdate(l2, itself);
138
139                     //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
140                         ↪ tion>(scope.TempTransactionLogFilename);
141
142                     l2 = links.Update(l2, l1);
143
144                     links.Delete(l2);
145
146                     ExceptionThrower();
147
148                     transaction.Commit();
149                 }
150
151                 Global.Trash = links.Count();
152             }
153         }
154         catch
155         {
156             Assert.False(lastScope == null);
157
158             var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
159                 ↪ astScope.TempTransactionLogFilename);
160
161             Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
162                 ↪ transitions[0].After.IsNull());
163
164             lastScope.DeleteFiles();
165         }
166     }
167
168     [Fact]
169     public static void TransactionUserCodeErrorSomeDataSavedTest()
170     {
171         // User Code Error (Autoreverted), some data saved
172         var itself = _constants.Itself;
173
174         TempLinksTestScope lastScope = null;
175         try

```



```

171 {
172     ulong l1;
173     ulong l2;
174
175     using (var scope = new TempLinksTestScope(useLog: true))
176     {
177         var links = scope.Links;
178         l1 = links.CreateAndUpdate(itself, itself);
179         l2 = links.CreateAndUpdate(itself, itself);
180
181         l2 = links.Update(l2, l2, l1, l2);
182
183         links.CreateAndUpdate(l2, itself);
184         links.CreateAndUpdate(l2, itself);
185
186         links.Unsync.DisposeIfPossible();
187
188         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
189             ↪ scope.TempTransactionLogFilename);
190     }
191
192     using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
193         ↪ useLog: true))
194     {
195         var links = scope.Links;
196         var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
197         using (var transaction = transactionsLayer.BeginTransaction())
198         {
199             l2 = links.Update(l2, l1);
200
201             links.Delete(l2);
202
203             ExceptionThrower();
204
205             transaction.Commit();
206         }
207
208         Global.Trash = links.Count();
209     }
210 }
211 catch
212 {
213     Assert.False(lastScope == null);
214
215     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
216         ↪ Scope.TempTransactionLogFilename);
217
218     lastScope.DeleteFiles();
219 }
220 }
221
222 [Fact]
223 public static void TransactionCommit()
224 {
225     var itself = _constants.Itself;
226
227     var tempDatabaseFilename = Path.GetTempFileName();
228     var tempTransactionLogFilename = Path.GetTempFileName();
229
230     // Commit
231     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
232         ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
233     using (var links = new UInt64Links(memoryAdapter))
234     {
235         using (var transaction = memoryAdapter.BeginTransaction())
236         {
237             var l1 = links.CreateAndUpdate(itself, itself);
238             var l2 = links.CreateAndUpdate(itself, itself);
239
240             Global.Trash = links.Update(l2, l2, l1, l2);
241
242             links.Delete(l1);
243
244             transaction.Commit();
245         }
246
247         Global.Trash = links.Count();
248     }
249 }

```

```

246         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
    ↪ sactionLogFilename);
247     }
248
249     [Fact]
250     public static void TransactionDamage()
251     {
252         var itself = _constants.Itself;
253
254         var tempDatabaseFilename = Path.GetTempFileName();
255         var tempTransactionLogFilename = Path.GetTempFileName();
256
257         // Commit
258         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
    ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
259         using (var links = new UInt64Links(memoryAdapter))
260         {
261             using (var transaction = memoryAdapter.BeginTransaction())
262             {
263                 var l1 = links.CreateAndUpdate(itself, itself);
264                 var l2 = links.CreateAndUpdate(itself, itself);
265
266                 Global.Trash = links.Update(l2, l2, l1, l2);
267
268                 links.Delete(l1);
269
270                 transaction.Commit();
271             }
272
273             Global.Trash = links.Count();
274         }
275
276         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
    ↪ sactionLogFilename);
277
278         // Damage database
279
280         FileHelpers.WriteFirst(tempTransactionLogFilename, new
    ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
281
282         // Try load damaged database
283         try
284         {
285             // TODO: Fix
286             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
    ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
287             using (var links = new UInt64Links(memoryAdapter))
288             {
289                 Global.Trash = links.Count();
290             }
291         }
292         catch (NotSupportedException ex)
293         {
294             Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
    ↪ yet.");
295         }
296
297         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
    ↪ sactionLogFilename);
298
299         File.Delete(tempDatabaseFilename);
300         File.Delete(tempTransactionLogFilename);
301     }
302
303     [Fact]
304     public static void Bug1Test()
305     {
306         var tempDatabaseFilename = Path.GetTempFileName();
307         var tempTransactionLogFilename = Path.GetTempFileName();
308
309         var itself = _constants.Itself;
310
311         // User Code Error (Autoreverted), some data saved
312         try
313         {
314             ulong l1;
315             ulong l2;
316
317             using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))

```

```

318     using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
319         ↪ tempTransactionLogFilename))
320     using (var links = new UInt64Links(memoryAdapter))
321     {
322         l1 = links.CreateAndUpdate(itself, itself);
323         l2 = links.CreateAndUpdate(itself, itself);
324
325         l2 = links.Update(l2, l2, l1, l2);
326
327         links.CreateAndUpdate(l2, itself);
328         links.CreateAndUpdate(l2, itself);
329     }
330
331     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp_
332         ↪ TransactionLogFilename);
333
334     using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
335     using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
336         ↪ tempTransactionLogFilename))
337     using (var links = new UInt64Links(memoryAdapter))
338     {
339         using (var transaction = memoryAdapter.BeginTransaction())
340         {
341             l2 = links.Update(l2, l1);
342
343             links.Delete(l2);
344
345             ExceptionThrower();
346
347             transaction.Commit();
348         }
349
350         Global.Trash = links.Count();
351     }
352 }
353 catch
354 {
355     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp_
356         ↪ TransactionLogFilename);
357 }
358
359 File.Delete(tempDatabaseFilename);
360 File.Delete(tempTransactionLogFilename);
361 }
362
363 private static void ExceptionThrower() => throw new InvalidOperationException();
364
365 [Fact]
366 public static void PathsTest()
367 {
368     var source = _constants.SourcePart;
369     var target = _constants.TargetPart;
370
371     using (var scope = new TempLinksTestScope())
372     {
373         var links = scope.Links;
374         var l1 = links.CreatePoint();
375         var l2 = links.CreatePoint();
376
377         var r1 = links.GetByKeys(l1, source, target, source);
378         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
379     }
380 }
381
382 [Fact]
383 public static void RecursiveStringFormattingTest()
384 {
385     using (var scope = new TempLinksTestScope(useSequences: true))
386     {
387         var links = scope.Links;
388         var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
389
390         var a = links.CreatePoint();
391         var b = links.CreatePoint();
392         var c = links.CreatePoint();
393
394         var ab = links.GetOrCreate(a, b);
395         var cb = links.GetOrCreate(c, b);
396         var ac = links.GetOrCreate(a, c);

```

```

393     a = links.Update(a, c, b);
394     b = links.Update(b, a, c);
395     c = links.Update(c, a, b);
396
397
398     Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
399     Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
400     Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
401
402     Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
403         ↪ "(5:(4:5 (6:5 4)) 6)");
404     Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
405         ↪ "(6:(5:(4:5 6) 6) 4)");
406     Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
407         ↪ "(4:(5:4 (6:5 4)) 6)");
408
409     // TODO: Think how to build balanced syntax tree while formatting structure (eg.
410     ↪ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)"
411
412     Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
413         ↪ "{5}{5}{4}{6}");
414     Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
415         ↪ "{5}{6}{6}{4}");
416     Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
417         ↪ "{4}{5}{4}{6}");
418 }
419 }
420
421 private static void DefaultFormatter(StringBuilder sb, ulong link)
422 {
423     sb.Append(link.ToString());
424 }
425
426 #endregion
427
428 #region Performance
429
430 /*
431 public static void RunAllPerformanceTests()
432 {
433     try
434     {
435         links.TestLinksInSteps();
436     }
437     catch (Exception ex)
438     {
439         ex.WriteToConsole();
440     }
441
442     return;
443
444     try
445     {
446         //ThreadPool.SetMaxThreads(2, 2);
447
448         // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
449         ↪ результат
450         // Также это дополнительно помогает в отладке
451         // Увеличивает вероятность попадания информации в кэши
452         for (var i = 0; i < 10; i++)
453         {
454             //0 - 10 ГБ
455             //Каждые 100 МБ срез цифр
456
457             //links.TestGetSourceFunction();
458             //links.TestGetSourceFunctionInParallel();
459             //links.TestGetTargetFunction();
460             //links.TestGetTargetFunctionInParallel();
461             links.Create64BillionLinks();
462
463             links.TestRandomSearchFixed();
464             //links.Create64BillionLinksInParallel();
465             links.TestEachFunction();
466             //links.TestForeach();
467             //links.TestParallelForeach();
468         }
469
470         links.TestDeletionOfAllLinks();
471     }
472 }

```

```

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         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
578
579         // Удаляем связь, из которой производилось считывание
580         links.Delete(firstLink);
581
582         ConsoleHelpers.Debug(
583             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
584             ↪ second), counter result: {3}",
585             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
586     }
587
588 [Fact(Skip = "performance test")]
589 public static void GetSourceInParallel()
590 {
591     using (var scope = new TempLinksTestScope())
592     {
593         var links = scope.Links;
594         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
595             ↪ parallel.", Iterations);
596
597         long counter = 0;
598
599         //var firstLink = links.First();
600         var firstLink = links.Create();
601
602         var sw = Stopwatch.StartNew();
603
604         // Тестируем саму функцию
605         Parallel.For(0, Iterations, x =>
606         {
607             Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
608             //Interlocked.Increment(ref counter);
609         });
610
611         var elapsedTime = sw.Elapsed;
612         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
613

```

```

614         links.Delete(firstLink);
615
616         ConsoleHelpers.Debug(
617             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
618     }
619 }
620
621 [Fact(Skip = "performance test")]
622 public static void TestGetTarget()
623 {
624     using (var scope = new TempLinksTestScope())
625     {
626         var links = scope.Links;
627         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
628             ↳ Iterations);
629
630         ulong counter = 0;
631
632         //var firstLink = links.First();
633         var firstLink = links.Create();
634
635         var sw = Stopwatch.StartNew();
636
637         for (ulong i = 0; i < Iterations; i++)
638         {
639             counter += links.GetTarget(firstLink);
640         }
641
642         var elapsedTime = sw.Elapsed;
643
644         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
645
646         links.Delete(firstLink);
647
648         ConsoleHelpers.Debug(
649             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
650     }
651 }
652
653 [Fact(Skip = "performance test")]
654 public static void TestGetTargetInParallel()
655 {
656     using (var scope = new TempLinksTestScope())
657     {
658         var links = scope.Links;
659         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
660             ↳ parallel.", Iterations);
661
662         long counter = 0;
663
664         //var firstLink = links.First();
665         var firstLink = links.Create();
666
667         var sw = Stopwatch.StartNew();
668
669         Parallel.For(0, Iterations, x =>
670         {
671             Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
672             //Interlocked.Increment(ref counter);
673         });
674
675         var elapsedTime = sw.Elapsed;
676
677         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
678
679         links.Delete(firstLink);
680
681         ConsoleHelpers.Debug(
682             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
683     }
684 }
685
686 // TODO: Заполнить базу данных перед тестом
687

```

```

688     /*
689     [Fact]
690     public void TestRandomSearchFixed()
691     {
692         var tempFilename = Path.GetTempFileName();
693
694         using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪ DefaultLinksSizeStep))
695         {
696             long iterations = 64 * 1024 * 1024 /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
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

```



```

759 public static void TestEach()
760 {
761     using (var scope = new TempLinksTestScope())
762     {
763         var links = scope.Links;
764
765         var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
766
767         ConsoleHelpers.Debug("Testing Each function.");
768
769         var sw = Stopwatch.StartNew();
770
771         links.Each(counter.IncrementAndReturnTrue);
772
773         var elapsedTime = sw.Elapsed;
774
775         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
776
777         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
↪         ↪ links per second)",
↪         ↪ counter, elapsedTime, (long)linksPerSecond);
778     }
779 }
780
781 /*
782 [Fact]
783 public static void TestForeach()
784 {
785     var tempFilename = Path.GetTempFileName();
786
787     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪     ↪ DefaultLinksSizeStep))
788     {
789         ulong counter = 0;
790
791         ConsoleHelpers.Debug("Testing foreach through links.");
792
793         var sw = Stopwatch.StartNew();
794
795         //foreach (var link in links)
796         //{
797             //    counter++;
798         //}
799
800         var elapsedTime = sw.Elapsed;
801
802         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
803
804         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
↪     ↪ links per second)", counter, elapsedTime, (long)linksPerSecond);
805     }
806
807     File.Delete(tempFilename);
808 }
809 */
810
811 /*
812 [Fact]
813 public static void TestParallelForeach()
814 {
815     var tempFilename = Path.GetTempFileName();
816
817     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪     ↪ DefaultLinksSizeStep))
818     {
819         long counter = 0;
820
821         ConsoleHelpers.Debug("Testing parallel foreach through links.");
822
823         var sw = Stopwatch.StartNew();
824
825         //Parallel.ForEach((IEnumerable<ulong>)links, x =>
826         //{
827             //    Interlocked.Increment(ref counter);
828         //});
829
830         var elapsedTime = sw.Elapsed;
831
832         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
833
834

```

```

835         ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
836         ↪ {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 / UInt64UnitedMemoryLinks.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)",
869         ↪ linksCreated, elapsedTime,
870         (long)linksPerSecond);
871     }
872
873 [Fact(Skip = "performance test")]
874 public static void Create64BillionLinksInParallel()
875 {
876     using (var scope = new TempLinksTestScope())
877     {
878         var links = scope.Links;
879         var linksBeforeTest = links.Count();
880
881         var sw = Stopwatch.StartNew();
882
883         long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;
884
885         ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
886
887         Parallel.For(0, linksToCreate, x => links.Create());
888
889         var elapsedTime = sw.Elapsed;
890
891         var linksCreated = links.Count() - linksBeforeTest;
892         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
893
894         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
895         ↪ linksCreated, elapsedTime,
896         (long)linksPerSecond);
897     }
898
899 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
900 public static void TestDeletionOfAllLinks()
901 {
902     using (var scope = new TempLinksTestScope())
903     {
904         var links = scope.Links;
905         var linksBeforeTest = links.Count();
906
907         ConsoleHelpers.Debug("Deleting all links");
908
909         var elapsedTime = Performance.Measure(links.DeleteAll);
910
911         var linksDeleted = linksBeforeTest - links.Count();

```

```

912         var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
913
914         ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
915             ↪ linksDeleted, elapsedTime,
916             (long)linksPerSecond);
917     }
918 }
919 #endregion
920 }
921 }

```

1.126 ./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.Memory.United.Generic;
15
16 namespace Platform.Data.Doublets.Tests
17 {
18     public static class UnicodeConvertersTests
19     {

```

```

20 [Fact]
21 public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
22 {
23     using (var scope = new TempLinksTestScope())
24     {
25         var links = scope.Links;
26         var meaningRoot = links.CreatePoint();
27         var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
28         var powerOf2ToUnaryNumberConverter = new
29             ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
30         var addressToUnaryNumberConverter = new
31             ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
32         var unaryNumberToAddressConverter = new
33             ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
34             ↪ powerOf2ToUnaryNumberConverter);
35         TestCharAndUnicodeSymbolConverters(links, meaningRoot,
36             ↪ addressToUnaryNumberConverter, unaryNumberToAddressConverter);
37     }
38 }
39
40 [Fact]
41 public static void CharAndRawNumberUnicodeSymbolConvertersTest()
42 {
43     using (var scope = new Scope<Types<HeapResizableDirectMemory,
44         ↪ UnitedMemoryLinks<ulong>>>())
45     {
46         var links = scope.Use<ILinks<ulong>>();
47         var meaningRoot = links.CreatePoint();
48         var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
49         var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
50         TestCharAndUnicodeSymbolConverters(links, meaningRoot,
51             ↪ addressToRawNumberConverter, rawNumberToAddressConverter);
52     }
53 }
54
55 private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
56     ↪ meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
57     ↪ numberToAddressConverter)
58 {
59     var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
60     var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
61         ↪ addressToNumberConverter, unicodeSymbolMarker);
62     var originalCharacter = 'H';
63     var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
64     var unicodeSymbolCriterionMatcher = new UnicodeSymbolCriterionMatcher<ulong>(links,
65         ↪ unicodeSymbolMarker);
66     var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
67         ↪ numberToAddressConverter, unicodeSymbolCriterionMatcher);
68     var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
69     Assert.Equal(originalCharacter, resultingCharacter);
70 }
71
72 [Fact]
73 public static void StringAndUnicodeSequenceConvertersTest()
74 {
75     using (var scope = new TempLinksTestScope())
76     {
77         var links = scope.Links;
78         var itself = links.Constants.Itself;
79
80         var meaningRoot = links.CreatePoint();
81         var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
82         var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
83         var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
84         var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
85         var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
86
87         var powerOf2ToUnaryNumberConverter = new
88             ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
89         var addressToUnaryNumberConverter = new
90             ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
91         var charToUnicodeSymbolConverter = new
92             ↪ CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
93             ↪ unicodeSymbolMarker);
94     }
95 }

```

```

80     var unaryNumberToAddressConverter = new
      ↳ UnaryNumberToAddressOrOperationConverter<ulong>(links,
      ↳ powerOf2ToUnaryNumberConverter);
81     var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
82     var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
      ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
83     var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
      ↳ frequencyPropertyMarker, frequencyMarker);
84     var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
      ↳ frequencyPropertyOperator, frequencyIncrementer);
85     var linkToItsFrequencyNumberConverter = new
      ↳ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
      ↳ unaryNumberToAddressConverter);
86     var sequenceToItsLocalElementLevelsConverter = new
      ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
      ↳ linkToItsFrequencyNumberConverter);
87     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
      ↳ sequenceToItsLocalElementLevelsConverter);
88
89     var stringToUnicodeSequenceConverter = new
      ↳ StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
      ↳ index, optimalVariantConverter, unicodeSequenceMarker);
90
91     var originalString = "Hello";
92
93     var unicodeSequenceLink =
      ↳ stringToUnicodeSequenceConverter.Convert(originalString);
94
95     var unicodeSymbolCriterionMatcher = new
      ↳ UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
96     var unicodeSymbolToCharConverter = new
      ↳ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
      ↳ unicodeSymbolCriterionMatcher);
97
98     var unicodeSequenceCriterionMatcher = new
      ↳ UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
99
100    var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
      ↳ unicodeSymbolCriterionMatcher.IsMatched);
101
102    var unicodeSequenceToStringConverter = new
      ↳ UnicodeSequenceToStringConverter<ulong>(links,
      ↳ unicodeSequenceCriterionMatcher, sequenceWalker,
      ↳ unicodeSymbolToCharConverter);
103
104    var resultingString =
      ↳ unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
105
106    Assert.Equal(originalString, resultingString);
107
108    }
109 }
110 }

```

Index

[./csharp/Platform.Data.Doublets.Tests/ComparisonTests.cs](#), 171
[./csharp/Platform.Data.Doublets.Tests/EqualityTests.cs](#), 171
[./csharp/Platform.Data.Doublets.Tests/GenericLinksTests.cs](#), 173
[./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs](#), 173
[./csharp/Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs](#), 174
[./csharp/Platform.Data.Doublets.Tests/ReadSequenceTests.cs](#), 177
[./csharp/Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs](#), 178
[./csharp/Platform.Data.Doublets.Tests/ScopeTests.cs](#), 179
[./csharp/Platform.Data.Doublets.Tests/SequencesTests.cs](#), 179
[./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs](#), 194
[./csharp/Platform.Data.Doublets.Tests/TempLinksTestScope.cs](#), 195
[./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs](#), 196
[./csharp/Platform.Data.Doublets.Tests/UInt64LinksTests.cs](#), 198
[./csharp/Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs](#), 211
[./csharp/Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs](#), 211
[./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/LinksHeader.cs](#), 30
[./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSizeBalancedTreeMethodsBase.cs](#), 31
[./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSourcesSizeBalancedTreeMethods.cs](#), 34
[./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsSizeBalancedTreeMethods.cs](#), 35
[./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSizeBalancedTreeMethodsBase.cs](#), 36
[./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesSizeBalancedTreeMethods.cs](#), 38
[./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsSizeBalancedTreeMethods.cs](#), 39
[./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinks.cs](#), 40
[./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinksBase.cs](#), 41
[./csharp/Platform.Data.Doublets/Memory/Split/Generic/UnusedLinksListMethods.cs](#), 51
[./csharp/Platform.Data.Doublets/Memory/Split/RawLinkDataPart.cs](#), 52
[./csharp/Platform.Data.Doublets/Memory/Split/RawLinkIndexPart.cs](#), 52
[./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksAvlBalancedTreeMethodsBase.cs](#), 53
[./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSizeBalancedTreeMethodsBase.cs](#), 57
[./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesAvlBalancedTreeMethods.cs](#), 60
[./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesSizeBalancedTreeMethods.cs](#), 62
[./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsAvlBalancedTreeMethods.cs](#), 62
[./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsSizeBalancedTreeMethods.cs](#), 64
[./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinks.cs](#), 65
[./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinksBase.cs](#), 66
[./csharp/Platform.Data.Doublets/Memory/United/Generic/UnusedLinksListMethods.cs](#), 73
[./csharp/Platform.Data.Doublets/Memory/United/RawLink.cs](#), 74
[./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs](#), 75
[./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs](#), 76
[./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs](#), 77

./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.cs, 79
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs, 80
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs, 81
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnitedMemoryLinks.cs, 82
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnusedLinksListMethods.cs, 83
./csharp/Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs, 84
./csharp/Platform.Data.Doublets/Numbers/Unary/LinkToltsFrequencyNumberConveter.cs, 84
./csharp/Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs, 85
./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs, 86
./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs, 87
./csharp/Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 88
./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 88
./csharp/Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 89
./csharp/Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 90
./csharp/Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 93
./csharp/Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 93
./csharp/Platform.Data.Doublets/Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 95
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs, 96
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs, 96
./csharp/Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 96
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 97
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 97
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 100
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 102
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToltsFrequencyValueConverter.cs, 102
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 102
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 103
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 104
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 104
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 104
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 105
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 106
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 106
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 107
./csharp/Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 107
./csharp/Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 108
./csharp/Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 108
./csharp/Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 109
./csharp/Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 109
./csharp/Platform.Data.Doublets/Sequences/Indexes/Unindex.cs, 110
./csharp/Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 110
./csharp/Platform.Data.Doublets/Sequences/Sequences.cs, 137
./csharp/Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 148
./csharp/Platform.Data.Doublets/Sequences/SequencesOptions.cs, 149
./csharp/Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 151
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 152
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 152
./csharp/Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 154
./csharp/Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 154
./csharp/Platform.Data.Doublets/Stacks/Stack.cs, 155
./csharp/Platform.Data.Doublets/Stacks/StackExtensions.cs, 156
./csharp/Platform.Data.Doublets/SynchronizedLinks.cs, 156
./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs, 158
./csharp/Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 159
./csharp/Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 165
./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 166
./csharp/Platform.Data.Doublets/Unicode/UnicodeMap.cs, 166
./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs, 169
./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 169
./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs, 170
./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 170