

## LinksPlatform's Platform.Data.Doublets Class Library

### 1.1 ./csharp/Platform.Data.Doublets/CriterionMatchers/TargetMatcher.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.CriterionMatchers
8 {
9     public class TargetMatcher<TLink> : LinksOperatorBase<TLink>, ICriterionMatcher<TLink>
10    {
11        private static readonly EqualityComparer<TLink> _equalityComparer =
12            ↳ EqualityComparer<TLink>.Default;
13
14        private readonly TLink _targetToMatch;
15
16        [MethodImpl(MethodImplOptions.AggressiveInlining)]
17        public TargetMatcher(ILinks<TLink> links, TLink targetToMatch) : base(links) =>
18            ↳ _targetToMatch = targetToMatch;
19
20        [MethodImpl(MethodImplOptions.AggressiveInlining)]
21        public bool IsMatched(TLink link) => _equalityComparer.Equals(_links.GetTarget(link),
22            ↳ _targetToMatch);
23    }
24 }
```

### 1.2 ./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    }
21 }
```

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

#### 1.4 ./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                 _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     }
60 }

```

#### 1.5 ./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

```

```

17     {
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         get => true;
20     }
21 }
22
23 protected readonly DisposableWithMultipleCallsAllowed Disposable;
24
25 [MethodImpl(MethodImplOptions.AggressiveInlining)]
26 protected LinksDisposableDecoratorBase(ILinks<TLink> links) : base(links) => Disposable
27     => = new DisposableWithMultipleCallsAllowed(Dispose);
28
29 [MethodImpl(MethodImplOptions.AggressiveInlining)]
30 ~LinksDisposableDecoratorBase() => Disposable.Destruct();
31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 public void Dispose() => Disposable.Dispose();
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 protected virtual void Dispose(bool manual, bool wasDisposed)
37 {
38     if (!wasDisposed)
39     {
40         _links.DisposeIfPossible();
41     }
42 }
43 }

```

#### 1.6 ./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     }
43 }

```

#### 1.7 ./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));
35    }
36 }

```

#### 1.8 ./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 }

```

#### 1.9 ./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));
18     }
19 }

```

#### 1.10 ./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;
21             var newLinkAddress = links.SearchOrDefault(substitution[constants.SourcePart],
22             ↪ substitution[constants.TargetPart]);
23             if (_equalityComparer.Equals(newLinkAddress, default))
24             {
25                 return links.Update(restrictions, substitution);
26             }
27             return ResolveAddressChangeConflict(restrictions[constants.IndexPart],
28             ↪ newLinkAddress);
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
33             ↪ newLinkAddress)
34         {
35             if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
36             ↪ _links.Exists(oldLinkAddress))
37             {
38                 _facade.Delete(oldLinkAddress);
39             }
40             return newLinkAddress;
41         }
42     }
43 }

```

#### 1.11 ./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 }

```

### 1.12 ./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 }
```

### 1.13 ./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 }
```

### 1.14 ./csharp/Platform.Data.Doublets/Decorators/UInt32Links.cs

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using TLink = System.UInt32;
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 UInt32Links : LinksDisposableDecoratorBase<TLink>
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public UInt32Links(ILinks<TLink> links) : base(links) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public override TLink Create(IList<TLink> restrictions) => _links.CreatePoint();
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
19         {
20             var constants = _constants;
21             var indexPartConstant = constants.IndexPart;
22             var sourcePartConstant = constants.SourcePart;
23             var targetPartConstant = constants.TargetPart;
```

```

24     var nullConstant = constants.Null;
25     var itselfConstant = constants.Itself;
26     var existedLink = nullConstant;
27     var updatedLink = restrictions[indexPartConstant];
28     var newSource = substitution[sourcePartConstant];
29     var newTarget = substitution[targetPartConstant];
30     var links = _links;
31     if (newSource != itselfConstant && newTarget != itselfConstant)
32     {
33         existedLink = links.SearchOrDefault(newSource, newTarget);
34     }
35     if (existedLink == nullConstant)
36     {
37         var before = links.GetLink(updatedLink);
38         if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
39             ↪ newTarget)
40         {
41             links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
42                 ↪ newSource,
43                 newTarget == itselfConstant ? updatedLink :
44                     ↪ newTarget);
45         }
46         return updatedLink;
47     }
48     else
49     {
50         return _facade.MergeAndDelete(updatedLink, existedLink);
51     }
52 }
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 public override void Delete(IList<TLink> restrictions)
56 {
57     var linkIndex = restrictions[_constants.IndexPart];
58     var links = _links;
59     links.EnforceResetValues(linkIndex);
60     _facade.DeleteAllUsages(linkIndex);
61     links.Delete(linkIndex);
62 }
63 }

```

### 1.15 ./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>

```

```

28 {
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public UInt64Links(ILinks<ulong> links) : base(links) { }
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     public override ulong Create(IList<ulong> restrictions) => _links.CreatePoint();
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
37     {
38         var constants = _constants;
39         var indexPartConstant = constants.IndexPart;
40         var sourcePartConstant = constants.SourcePart;
41         var targetPartConstant = constants.TargetPart;
42         var nullConstant = constants.Null;
43         var itselfConstant = constants.Itself;
44         var existedLink = nullConstant;
45         var updatedLink = restrictions[indexPartConstant];
46         var newSource = substitution[sourcePartConstant];
47         var newTarget = substitution[targetPartConstant];
48         var links = _links;
49         if (newSource != itselfConstant && newTarget != itselfConstant)
50         {
51             existedLink = links.SearchOrDefault(newSource, newTarget);
52         }
53         if (existedLink == nullConstant)
54         {
55             var before = links.GetLink(updatedLink);
56             if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
57                 ↪ newTarget)
58             {
59                 links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
60                     ↪ newSource,
61                                     newTarget == itselfConstant ? updatedLink :
62                     ↪ newTarget);
63             }
64             return updatedLink;
65         }
66         else
67         {
68             return _facade.MergeAndDelete(updatedLink, existedLink);
69         }
70     }
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     public override void Delete(IList<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 }

```

## 1.16 ./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     {

```



```

20 private static readonly EqualityComparer<TLink> _equalityComparer =
    ↳ EqualityComparer<TLink>.Default;
21
22 public UniLinks(ILinks<TLink> links) : base(links) { }
23
24 private struct Transition
25 {
26     public IList<TLink> Before;
27     public IList<TLink> After;
28
29     public Transition(IList<TLink> before, IList<TLink> after)
30     {
31         Before = before;
32         After = after;
33     }
34 }
35
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     /////            }
89     /////            if (!Memory.Each(handler, restriction))
90     /////            {
91     /////                return Constants.Break;
92     /////            }
93     /////        }
94     /////    }
95     /////}
96     /////else
97     /////{
98     /////}
99 }

```

```

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

```

```

163         //     }
164         //     if (!matchedLinks.IsNullOrEmpty())
165         //     {
166             var totalMatchedLinks = matchedLinks.Count;
167             for (var i = 0; i < totalMatchedLinks; i++)
168             {
169                 var matchedLink = matchedLinks[i];
170                 if (substitutedHandler != null)
171                 {
172                     var newValue = new List<T>(); // TODO: Prepare value to update here
173                     // TODO: Decide is it actually needed to use Before and After
174                     ↪ substitution handling.
175                     var substitutedDecision = substitutedHandler(matchedLink,
176                     ↪ newValue);
177                     if (Equals(substitutedDecision, Constants.Break))
178                         return Constants.Break;
179                     if (Equals(substitutedDecision, Constants.Continue))
180                     {
181                         // Actual update here
182                         Memory.SetLinkValue(newValue);
183                     }
184                     if (Equals(substitutedDecision, Constants.Skip))
185                     {
186                         // Cancel the update. TODO: decide use separate Cancel
187                         ↪ constant or Skip is enough?
188                     }
189                 }
190             }
191         }
192     }
193     // }
194     // }
195     // }
196     // }
197     // }
198     // }
199     // }
200     // }
201     // }
202     // }
203     // }
204     // }
205     // }
206     // }
207     // }
208     // }
209     // }
210     // }
211     // }
212     // }
213     // }
214     // }
215     // }
216     // }
217     // }
218     // }
219     // }
220     // }
221     // }
222     // }
223     // }
224     // }
225     // }
226     // }
227     // }
228     // }
229     // }
230     // }
231     // }
232     // }
233     // }
234     // }
235     // }
236     // }
237     // }
238     // }
239     // }
240     // }
241     // }
242     // }
243     // }
244     // }
245     // }
246     // }
247     // }
248     // }
249     // }
250     // }
251     // }
252     // }
253     // }
254     // }
255     // }
256     // }
257     // }
258     // }
259     // }
260     // }
261     // }
262     // }
263     // }
264     // }
265     // }
266     // }
267     // }
268     // }
269     // }
270     // }
271     // }
272     // }
273     // }
274     // }
275     // }
276     // }
277     // }
278     // }
279     // }
280     // }
281     // }
282     // }
283     // }
284     // }
285     // }
286     // }
287     // }
288     // }
289     // }
290     // }
291     // }
292     // }
293     // }
294     // }
295     // }
296     // }
297     // }
298     // }
299     // }
300     // }
301     // }
302     // }
303     // }
304     // }
305     // }
306     // }
307     // }
308     // }
309     // }
310     // }
311     // }
312     // }
313     // }
314     // }
315     // }
316     // }
317     // }
318     // }
319     // }
320     // }
321     // }
322     // }
323     // }
324     // }
325     // }
326     // }
327     // }
328     // }
329     // }
330     // }
331     // }
332     // }
333     // }
334     // }
335     // }
336     // }
337     // }
338     // }
339     // }
340     // }
341     // }
342     // }
343     // }
344     // }
345     // }
346     // }
347     // }
348     // }
349     // }
350     // }
351     // }
352     // }
353     // }
354     // }
355     // }
356     // }
357     // }
358     // }
359     // }
360     // }
361     // }
362     // }
363     // }
364     // }
365     // }
366     // }
367     // }
368     // }
369     // }
370     // }
371     // }
372     // }
373     // }
374     // }
375     // }
376     // }
377     // }
378     // }
379     // }
380     // }
381     // }
382     // }
383     // }
384     // }
385     // }
386     // }
387     // }
388     // }
389     // }
390     // }
391     // }
392     // }
393     // }
394     // }
395     // }
396     // }
397     // }
398     // }
399     // }
400     // }
401     // }
402     // }
403     // }
404     // }
405     // }
406     // }
407     // }
408     // }
409     // }
410     // }
411     // }
412     // }
413     // }
414     // }
415     // }
416     // }
417     // }
418     // }
419     // }
420     // }
421     // }
422     // }
423     // }
424     // }
425     // }
426     // }
427     // }
428     // }
429     // }
430     // }
431     // }
432     // }
433     // }
434     // }
435     // }
436     // }
437     // }
438     // }
439     // }
440     // }
441     // }
442     // }
443     // }
444     // }
445     // }
446     // }
447     // }
448     // }
449     // }
450     // }
451     // }
452     // }
453     // }
454     // }
455     // }
456     // }
457     // }
458     // }
459     // }
460     // }
461     // }
462     // }
463     // }
464     // }
465     // }
466     // }
467     // }
468     // }
469     // }
470     // }
471     // }
472     // }
473     // }
474     // }
475     // }
476     // }
477     // }
478     // }
479     // }
480     // }
481     // }
482     // }
483     // }
484     // }
485     // }
486     // }
487     // }
488     // }
489     // }
490     // }
491     // }
492     // }
493     // }
494     // }
495     // }
496     // }
497     // }
498     // }
499     // }
500     // }
501     // }
502     // }
503     // }
504     // }
505     // }
506     // }
507     // }
508     // }
509     // }
510     // }
511     // }
512     // }
513     // }
514     // }
515     // }
516     // }
517     // }
518     // }
519     // }
520     // }
521     // }
522     // }
523     // }
524     // }
525     // }
526     // }
527     // }
528     // }
529     // }
530     // }
531     // }
532     // }
533     // }
534     // }
535     // }
536     // }
537     // }
538     // }
539     // }
540     // }
541     // }
542     // }
543     // }
544     // }
545     // }
546     // }
547     // }
548     // }
549     // }
550     // }
551     // }
552     // }
553     // }
554     // }
555     // }
556     // }
557     // }
558     // }
559     // }
560     // }
561     // }
562     // }
563     // }
564     // }
565     // }
566     // }
567     // }
568     // }
569     // }
570     // }
571     // }
572     // }
573     // }
574     // }
575     // }
576     // }
577     // }
578     // }
579     // }
580     // }
581     // }
582     // }
583     // }
584     // }
585     // }
586     // }
587     // }
588     // }
589     // }
590     // }
591     // }
592     // }
593     // }
594     // }
595     // }
596     // }
597     // }
598     // }
599     // }
600     // }
601     // }
602     // }
603     // }
604     // }
605     // }
606     // }
607     // }
608     // }
609     // }
610     // }
611     // }
612     // }
613     // }
614     // }
615     // }
616     // }
617     // }
618     // }
619     // }
620     // }
621     // }
622     // }
623     // }
624     // }
625     // }
626     // }
627     // }
628     // }
629     // }
630     // }
631     // }
632     // }
633     // }
634     // }
635     // }
636     // }
637     // }
638     // }
639     // }
640     // }
641     // }
642     // }
643     // }
644     // }
645     // }
646     // }
647     // }
648     // }
649     // }
650     // }
651     // }
652     // }
653     // }
654     // }
655     // }
656     // }
657     // }
658     // }
659     // }
660     // }
661     // }
662     // }
663     // }
664     // }
665     // }
666     // }
667     // }
668     // }
669     // }
670     // }
671     // }
672     // }
673     // }
674     // }
675     // }
676     // }
677     // }
678     // }
679     // }
680     // }
681     // }
682     // }
683     // }
684     // }
685     // }
686     // }
687     // }
688     // }
689     // }
690     // }
691     // }
692     // }
693     // }
694     // }
695     // }
696     // }
697     // }
698     // }
699     // }
700     // }
701     // }
702     // }
703     // }
704     // }
705     // }
706     // }
707     // }
708     // }
709     // }
710     // }
711     // }
712     // }
713     // }
714     // }
715     // }
716     // }
717     // }
718     // }
719     // }
720     // }
721     // }
722     // }
723     // }
724     // }
725     // }
726     // }
727     // }
728     // }
729     // }
730     // }
731     // }
732     // }
733     // }
734     // }
735     // }
736     // }
737     // }
738     // }
739     // }
740     // }
741     // }
742     // }
743     // }
744     // }
745     // }
746     // }
747     // }
748     // }
749     // }
750     // }
751     // }
752     // }
753     // }
754     // }
755     // }
756     // }
757     // }
758     // }
759     // }
760     // }
761     // }
762     // }
763     // }
764     // }
765     // }
766     // }
767     // }
768     // }
769     // }
770     // }
771     // }
772     // }
773     // }
774     // }
775     // }
776     // }
777     // }
778     // }
779     // }
780     // }
781     // }
782     // }
783     // }
784     // }
785     // }
786     // }
787     // }
788     // }
789     // }
790     // }
791     // }
792     // }
793     // }
794     // }
795     // }
796     // }
797     // }
798     // }
799     // }
800     // }
801     // }
802     // }
803     // }
804     // }
805     // }
806     // }
807     // }
808     // }
809     // }
810     // }
811     // }
812     // }
813     // }
814     // }
815     // }
816     // }
817     // }
818     // }
819     // }
820     // }
821     // }
822     // }
823     // }
824     // }
825     // }
826     // }
827     // }
828     // }
829     // }
830     // }
831     // }
832     // }
833     // }
834     // }
835     // }
836     // }
837     // }
838     // }
839     // }
840     // }
841     // }
842     // }
843     // }
844     // }
845     // }
846     // }
847     // }
848     // }
849     // }
850     // }
851     // }
852     // }
853     // }
854     // }
855     // }
856     // }
857     // }
858     // }
859     // }
860     // }
861     // }
862     // }
863     // }
864     // }
865     // }
866     // }
867     // }
868     // }
869     // }
870     // }
871     // }
872     // }
873     // }
874     // }
875     // }
876     // }
877     // }
878     // }
879     // }
880     // }
881     // }
882     // }
883     // }
884     // }
885     // }
886     // }
887     // }
888     // }
889     // }
890     // }
891     // }
892     // }
893     // }
894     // }
895     // }
896     // }
897     // }
898     // }
899     // }
900     // }
901     // }
902     // }
903     // }
904     // }
905     // }
906     // }
907     // }
908     // }
909     // }
910     // }
911     // }
912     // }
913     // }
914     // }
915     // }
916     // }
917     // }
918     // }
919     // }
920     // }
921     // }
922     // }
923     // }
924     // }
925     // }
926     // }
927     // }
928     // }
929     // }
930     // }
931     // }
932     // }
933     // }
934     // }
935     // }
936     // }
937     // }
938     // }
939     // }
940     // }
941     // }
942     // }
943     // }
944     // }
945     // }
946     // }
947     // }
948     // }
949     // }
950     // }
951     // }
952     // }
953     // }
954     // }
955     // }
956     // }
957     // }
958     // }
959     // }
960     // }
961     // }
962     // }
963     // }
964     // }
965     // }
966     // }
967     // }
968     // }
969     // }
970     // }
971     // }
972     // }
973     // }
974     // }
975     // }
976     // }
977     // }
978     // }
979     // }
980     // }
981     // }
982     // }
983     // }
984     // }
985     // }
986     // }
987     // }
988     // }
989     // }
990     // }
991     // }
992     // }
993     // }
994     // }
995     // }
996     // }
997     // }
998     // }
999     // }
1000    // }

```

```

233         return substitutionHandler(before, after);
234     }
235     return constants.Continue;
236 }
237 else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
238 {
239     if (patternOrCondition.Count == 1)
240     {
241         var linkToDelete = patternOrCondition[0];
242         var before = _links.GetLink(linkToDelete);
243         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
244             ↪ constants.Break))
245         {
246             return constants.Break;
247         }
248         var after = Array.Empty<TLink>();
249         _links.Update(linkToDelete, constants.Null, constants.Null);
250         _links.Delete(linkToDelete);
251         if (matchHandler != null)
252         {
253             return substitutionHandler(before, after);
254         }
255         return constants.Continue;
256     }
257     else
258     {
259         throw new NotSupportedException();
260     }
261 }
262 else // Replace / Update
263 {
264     if (patternOrCondition.Count == 1) //-V3125
265     {
266         var linkToUpdate = patternOrCondition[0];
267         var before = _links.GetLink(linkToUpdate);
268         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
269             ↪ constants.Break))
270         {
271             return constants.Break;
272         }
273         var after = (IList<TLink>)substitution.ToArray(); //-V3125
274         if (_equalityComparer.Equals(after[0], default))
275         {
276             after[0] = linkToUpdate;
277         }
278         if (substitution.Count == 1)
279         {
280             if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
281             {
282                 after = _links.GetLink(substitution[0]);
283                 _links.Update(linkToUpdate, constants.Null, constants.Null);
284                 _links.Delete(linkToUpdate);
285             }
286         }
287         else if (substitution.Count == 3)
288         {
289             //Links.Update(after);
290         }
291         else
292         {
293             throw new NotSupportedException();
294         }
295         if (matchHandler != null)
296         {
297             return substitutionHandler(before, after);
298         }
299         return constants.Continue;
300     }
301     else
302     {
303         throw new NotSupportedException();
304     }
305 }
306 }
307
308 /// <remarks>
309 /// IList[IList[IList[T]]]
310 /// |         |         |         |||

```

```

309 /// | | | | |
310 /// | | | | | link | |
311 /// | | | | |
312 /// | | | | | change | |
313 /// | | | | |
314 /// | | | | | changes
315 /// </remarks>
316 public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink>
    ↳ substitution)
317 {
318     var changes = new List<IList<IList<TLink>>>();
319     var @continue = _constants.Continue;
320     Trigger(condition, AlwaysContinue, substitution, (before, after) =>
321     {
322         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 }

```

### 1.17 ./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 readonly T Source;
15         public readonly T Target;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public Doublet(T source, T target)
19         {
20             Source = source;
21             Target = target;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public override string ToString() => $"{Source}->{Target}";
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
29             ↳ && _equalityComparer.Equals(Target, other.Target);
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public override bool Equals(object obj) => obj is Doublet<T> doublet ?
33             ↳ base.Equals(doublet) : false;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public override int GetHashCode() => (Source, Target).GetHashCode();
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public static bool operator ==(Doublet<T> left, Doublet<T> right) => left.Equals(right);
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public static bool operator !=(Doublet<T> left, Doublet<T> right) => !(left == right);
43     }
44 }

```

### 1.18 ./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 }

```

### 1.19 ./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 }

```

### 1.20 ./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()));

```

```

45     var source =
46         ↪ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
47     var target =
48         ↪ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
49     links.SearchOrDefault(source, target);
50 }
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, ulong
53     ↪ amountOfDeletions)
54 {
55     var random = RandomHelpers.Default;
56     var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
57     var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
58     var linksCount = addressToUInt64Converter.Convert(links.Count());
59     var min = amountOfDeletions > linksCount ? OUL : linksCount - amountOfDeletions;
60     for (var i = OUL; i < amountOfDeletions; i++)
61     {
62         linksCount = addressToUInt64Converter.Convert(links.Count());
63         if (linksCount <= min)
64         {
65             break;
66         }
67         var linksAddressRange = new Range<ulong>(min, linksCount);
68         var link =
69             ↪ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
70         links.Delete(link);
71     }
72 }
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 public static void Delete<TLink>(this ILinks<TLink> links, TLink linkToDelete) =>
75     ↪ links.Delete(new LinkAddress<TLink>(linkToDelete));
76
77 /// <remarks>
78 /// TODO: Возможно есть очень простой способ это сделать.
79 /// (Например просто удалить файл, или изменить его размер таким образом,
80 /// чтобы удалился весь контент)
81 /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
82 /// </remarks>
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 public static void DeleteAll<TLink>(this ILinks<TLink> links)
85 {
86     var equalityComparer = EqualityComparer<TLink>.Default;
87     var comparer = Comparer<TLink>.Default;
88     for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
89         ↪ Arithmetic.Decrement(i))
90     {
91         links.Delete(i);
92         if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
93         {
94             i = links.Count();
95         }
96     }
97 }
98
99 [MethodImpl(MethodImplOptions.AggressiveInlining)]
100 public static TLink First<TLink>(this ILinks<TLink> links)
101 {
102     TLink firstLink = default;
103     var equalityComparer = EqualityComparer<TLink>.Default;
104     if (equalityComparer.Equals(links.Count(), default))
105     {
106         throw new InvalidOperationException("В хранилище нет связей.");
107     }
108     links.Each(links.Constants.Any, links.Constants.Any, link =>
109     {
110         firstLink = link[links.Constants.IndexPart];
111         return links.Constants.Break;
112     });
113     if (equalityComparer.Equals(firstLink, default))
114     {
115         throw new InvalidOperationException("В процессе поиска по хранилищу не было
116             ↪ найдено связей.");
117     }
118     return firstLink;
119 }

```

```

116 [MethodImpl(MethodImplOptions.AggressiveInlining)]
117 public static IList<TLink> SingleOrDefault<TLink>(this ILinks<TLink> links, IList<TLink>
118     ↪ query)
119 {
120     IList<TLink> result = null;
121     var count = 0;
122     var constants = links.Constants;
123     var @continue = constants.Continue;
124     var @break = constants.Break;
125     links.Each(linkHandler, query);
126     return result;
127
128     TLink linkHandler(IList<TLink> link)
129     {
130         if (count == 0)
131         {
132             result = link;
133             count++;
134             return @continue;
135         }
136         else
137         {
138             result = null;
139             return @break;
140         }
141     }
142 }
143
144 #region Paths
145
146 /// <remarks>
147 /// TODO: Как так? Как то что ниже может быть корректно?
148 /// Скорее всего практически не применимо
149 /// Предполагалось, что можно было конвертировать формируемый в проходе через
150     ↪ SequenceWalker
151 /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
152 /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
153 /// </remarks>
154 [MethodImpl(MethodImplOptions.AggressiveInlining)]
155 public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
156     ↪ path)
157 {
158     var current = path[0];
159     //EnsureLinkExists(current, "path");
160     if (!links.Exists(current))
161     {
162         return false;
163     }
164     var equalityComparer = EqualityComparer<TLink>.Default;
165     var constants = links.Constants;
166     for (var i = 1; i < path.Length; i++)
167     {
168         var next = path[i];
169         var values = links.GetLink(current);
170         var source = values[constants.SourcePart];
171         var target = values[constants.TargetPart];
172         if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
173             ↪ next))
174         {
175             //throw new InvalidOperationException(string.Format("Невозможно выбрать
176             ↪ путь, так как и Source и Target совпадают с элементом пути {0}.", next));
177             return false;
178         }
179         if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
180             ↪ target))
181         {
182             //throw new InvalidOperationException(string.Format("Невозможно продолжить
183             ↪ путь через элемент пути {0}", next));
184             return false;
185         }
186         current = next;
187     }
188     return true;
189 }
190
191 /// <remarks>
192 /// Может потребовать дополнительного стека для PathElement's при использовании
193     ↪ SequenceWalker.

```



```

187 /// </remarks>
188 [MethodImpl(MethodImplOptions.AggressiveInlining)]
189 public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink root, params int[]
    ↳ path)
190 {
191     links.EnsureLinkExists(root, "root");
192     var currentLink = root;
193     for (var i = 0; i < path.Length; i++)
194     {
195         currentLink = links.GetLink(currentLink)[path[i]];
196     }
197     return currentLink;
198 }
199
200 [MethodImpl(MethodImplOptions.AggressiveInlining)]
201 public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
    ↳ links, TLink root, ulong size, ulong index)
202 {
203     var constants = links.Constants;
204     var source = constants.SourcePart;
205     var target = constants.TargetPart;
206     if (!Platform.Numbers.Math.IsPowerOfTwo(size))
207     {
208         throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
            ↳ than powers of two are not supported.");
209     }
210     var path = new BitArray(BitConverter.GetBytes(index));
211     var length = Bit.GetLowestPosition(size);
212     links.EnsureLinkExists(root, "root");
213     var currentLink = root;
214     for (var i = length - 1; i >= 0; i--)
215     {
216         currentLink = links.GetLink(currentLink)[path[i] ? target : source];
217     }
218     return currentLink;
219 }
220
221 #endregion
222
223 /// <summary>
224 /// Возвращает индекс указанной связи.
225 /// </summary>
226 /// <param name="links">Хранилище связей.</param>
227 /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↳ содержимого.</param>
228 /// <returns>Индекс начальной связи для указанной связи.</returns>
229 [MethodImpl(MethodImplOptions.AggressiveInlining)]
230 public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↳ link[links.Constants.IndexPart];
231
232 /// <summary>
233 /// Возвращает индекс начальной (Source) связи для указанной связи.
234 /// </summary>
235 /// <param name="links">Хранилище связей.</param>
236 /// <param name="link">Индекс связи.</param>
237 /// <returns>Индекс начальной связи для указанной связи.</returns>
238 [MethodImpl(MethodImplOptions.AggressiveInlining)]
239 public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
    ↳ links.GetLink(link)[links.Constants.SourcePart];
240
241 /// <summary>
242 /// Возвращает индекс начальной (Source) связи для указанной связи.
243 /// </summary>
244 /// <param name="links">Хранилище связей.</param>
245 /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↳ содержимого.</param>
246 /// <returns>Индекс начальной связи для указанной связи.</returns>
247 [MethodImpl(MethodImplOptions.AggressiveInlining)]
248 public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↳ link[links.Constants.SourcePart];
249
250 /// <summary>
251 /// Возвращает индекс конечной (Target) связи для указанной связи.
252 /// </summary>
253 /// <param name="links">Хранилище связей.</param>
254 /// <param name="link">Индекс связи.</param>
255 /// <returns>Индекс конечной связи для указанной связи.</returns>
256 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

257 public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
258     ↳ links.GetLink(link)[links.Constants.TargetPart];
259
260 /// <summary>
261 /// Возвращает индекс конечной (Target) связи для указанной связи.
262 /// </summary>
263 /// <param name="links">Хранилище связей.</param>
264 /// <param name="link">Связь представленная списком, состоящим из её адреса и
265     ↳ содержимого.</param>
266 /// <returns>Индекс конечной связи для указанной связи.</returns>
267 [MethodImpl(MethodImplOptions.AggressiveInlining)]
268 public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
269     ↳ link[links.Constants.TargetPart];
270
271 /// <summary>
272 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
273     ↳ (handler) для каждой подходящей связи.
274 /// </summary>
275 /// <param name="links">Хранилище связей.</param>
276 /// <param name="handler">Обработчик каждой подходящей связи.</param>
277 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
278     ↳ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
279     ↳ Any - отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
280 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
281     ↳ случае.</returns>
282 [MethodImpl(MethodImplOptions.AggressiveInlining)]
283 public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
284     ↳ handler, params TLink[] restrictions)
285     => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
286     ↳ links.Constants.Continue);
287
288 /// <summary>
289 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
290     ↳ (handler) для каждой подходящей связи.
291 /// </summary>
292 /// <param name="links">Хранилище связей.</param>
293 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
294     ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
295     ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
296 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
297     ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
298     ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
299 /// <param name="handler">Обработчик каждой подходящей связи.</param>
300 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
301     ↳ случае.</returns>
302 [MethodImpl(MethodImplOptions.AggressiveInlining)]
303 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
304     ↳ Func<TLink, bool> handler)
305 {
306     var constants = links.Constants;
307     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
308     ↳ constants.Break, constants.Any, source, target);
309 }
310
311 /// <summary>
312 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
313     ↳ (handler) для каждой подходящей связи.
314 /// </summary>
315 /// <param name="links">Хранилище связей.</param>
316 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
317     ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
318     ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
319 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
320     ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
321     ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
322 /// <param name="handler">Обработчик каждой подходящей связи.</param>
323 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
324     ↳ случае.</returns>
325 [MethodImpl(MethodImplOptions.AggressiveInlining)]
326 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
327     ↳ Func<IList<TLink>, TLink> handler) => links.Each(handler, links.Constants.Any,
328     ↳ source, target);
329
330 [MethodImpl(MethodImplOptions.AggressiveInlining)]
331 public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
332     ↳ restrictions)

```

```

307 {
308     var arraySize = CheckedConverter<TLink,
        ↳ ulong>.Default.Convert(links.Count(restrictions));
309     if (arraySize > 0)
310     {
311         var array = new IList<TLink>[arraySize];
312         var filler = new ArrayFiller<IList<TLink>, TLink>(array,
        ↳ links.Constants.Continue);
313         links.Each(filler.AddAndReturnConstant, restrictions);
314         return array;
315     }
316     else
317     {
318         return Array.Empty<IList<TLink>>();
319     }
320 }
321
322 [MethodImpl(MethodImplOptions.AggressiveInlining)]
323 public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
        ↳ restrictions)
324 {
325     var arraySize = CheckedConverter<TLink,
        ↳ ulong>.Default.Convert(links.Count(restrictions));
326     if (arraySize > 0)
327     {
328         var array = new TLink[arraySize];
329         var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
330         links.Each(filler.AddFirstAndReturnConstant, restrictions);
331         return array;
332     }
333     else
334     {
335         return Array.Empty<TLink>();
336     }
337 }
338
339 /// <summary>
340 /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
        ↳ в хранилище связей.
341 /// </summary>
342 /// <param name="links">Хранилище связей.</param>
343 /// <param name="source">Начало связи.</param>
344 /// <param name="target">Конец связи.</param>
345 /// <returns>Значение, определяющее существует ли связь.</returns>
346 [MethodImpl(MethodImplOptions.AggressiveInlining)]
347 public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
        ↳ => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
        ↳ default) > 0;
348
349 #region Ensure
350 // TODO: May be move to EnsureExtensions or make it both there and here
351
352 [MethodImpl(MethodImplOptions.AggressiveInlining)]
353 public static void EnsureLinkExists<TLink>(this ILinks<TLink> links, IList<TLink>
        ↳ restrictions)
354 {
355     for (var i = 0; i < restrictions.Count; i++)
356     {
357         if (!links.Exists(restrictions[i]))
358         {
359             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
        ↳ $"sequence[{i}]");
360         }
361     }
362 }
363
364 [MethodImpl(MethodImplOptions.AggressiveInlining)]
365 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
        ↳ reference, string argumentName)
366 {
367     if (links.Constants.IsInternalReference(reference) && !links.Exists(reference))
368     {
369         throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
370     }
371 }
372
373 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

374 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
    ↳ IList<TLink> restrictions, string argumentName)
375 {
376     for (int i = 0; i < restrictions.Count; i++)
377     {
378         links.EnsureInnerReferenceExists(restrictions[i], argumentName);
379     }
380 }
381
382 [MethodImpl(MethodImplOptions.AggressiveInlining)]
383 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
    ↳ restrictions)
384 {
385     var equalityComparer = EqualityComparer<TLink>.Default;
386     var any = links.Constants.Any;
387     for (var i = 0; i < restrictions.Count; i++)
388     {
389         if (!equalityComparer.Equals(restrictions[i], any) &&
    ↳ !links.Exists(restrictions[i]))
390         {
391             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
    ↳ $"{sequence[{i}]"}");
392         }
393     }
394 }
395
396 [MethodImpl(MethodImplOptions.AggressiveInlining)]
397 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
    ↳ string argumentName)
398 {
399     var equalityComparer = EqualityComparer<TLink>.Default;
400     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
401     {
402         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
403     }
404 }
405
406 [MethodImpl(MethodImplOptions.AggressiveInlining)]
407 public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
    ↳ link, string argumentName)
408 {
409     var equalityComparer = EqualityComparer<TLink>.Default;
410     if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
411     {
412         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
413     }
414 }
415
416 /// <param name="links">Хранилище связей.</param>
417 [MethodImpl(MethodImplOptions.AggressiveInlining)]
418 public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
    ↳ TLink target)
419 {
420     if (links.Exists(source, target))
421     {
422         throw new LinkWithSameValueAlreadyExistsException();
423     }
424 }
425
426 /// <param name="links">Хранилище связей.</param>
427 [MethodImpl(MethodImplOptions.AggressiveInlining)]
428 public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
429 {
430     if (links.HasUsages(link))
431     {
432         throw new ArgumentLinkHasDependenciesException<TLink>(link);
433     }
434 }
435
436 /// <param name="links">Хранилище связей.</param>
437 [MethodImpl(MethodImplOptions.AggressiveInlining)]
438 public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ addresses) => links.EnsureCreated(links.Create, addresses);
439
440 /// <param name="links">Хранилище связей.</param>
441 [MethodImpl(MethodImplOptions.AggressiveInlining)]
442 public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ addresses) => links.EnsureCreated(links.CreatePoint, addresses);

```

```

443
444 /// <param name="links">Хранилище связей.</param>
445 [MethodImpl(MethodImplOptions.AggressiveInlining)]
446 public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
    ↪ params TLink[] addresses)
447 {
448     var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
449     var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
450     var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
    ↪ !links.Exists(x)));
451     if (nonExistentAddresses.Count > 0)
452     {
453         var max = nonExistentAddresses.Max();
454         max = uInt64ToAddressConverter.Convert(System.Math.Min(addressToUInt64Converter.
    ↪ Convert(max),
    ↪ addressToUInt64Converter.Convert(links.Constants.InternalReferencesRange.Max
    ↪ imum)));
455         var createdLinks = new List<TLink>();
456         var equalityComparer = EqualityComparer<TLink>.Default;
457         TLink createdLink = creator();
458         while (!equalityComparer.Equals(createdLink, max))
459         {
460             createdLinks.Add(createdLink);
461         }
462         for (var i = 0; i < createdLinks.Count; i++)
463         {
464             if (!nonExistentAddresses.Contains(createdLinks[i]))
465             {
466                 links.Delete(createdLinks[i]);
467             }
468         }
469     }
470 }
471
472 #endregion
473
474 /// <param name="links">Хранилище связей.</param>
475 [MethodImpl(MethodImplOptions.AggressiveInlining)]
476 public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
477 {
478     var constants = links.Constants;
479     var values = links.GetLink(link);
480     TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link,
    ↪ constants.Any));
481     var equalityComparer = EqualityComparer<TLink>.Default;
482     if (equalityComparer.Equals(values[constants.SourcePart], link))
483     {
484         usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
485     }
486     TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
    ↪ link));
487     if (equalityComparer.Equals(values[constants.TargetPart], link))
488     {
489         usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
490     }
491     return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
492 }
493
494 /// <param name="links">Хранилище связей.</param>
495 [MethodImpl(MethodImplOptions.AggressiveInlining)]
496 public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
    ↪ Comparer<TLink>.Default.Compare(links.CountUsages(link), default) > 0;
497
498 /// <param name="links">Хранилище связей.</param>
499 [MethodImpl(MethodImplOptions.AggressiveInlining)]
500 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
    ↪ TLink target)
501 {
502     var constants = links.Constants;
503     var values = links.GetLink(link);
504     var equalityComparer = EqualityComparer<TLink>.Default;
505     return equalityComparer.Equals(values[constants.SourcePart], source) &&
    ↪ equalityComparer.Equals(values[constants.TargetPart], target);
506 }
507
508 /// <summary>
509 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
510 /// </summary>

```

```

511 /// <param name="links">Хранилище связей.</param>
512 /// <param name="source">Индекс связи, которая является началом для искомой
513 → связи.</param>
514 /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
515 /// <returns>Индекс искомой связи с указанными Source (началом) и Target
516 → (концом).</returns>
517 [MethodImpl(MethodImplOptions.AggressiveInlining)]
518 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
519 → target)
520 {
521     var constants = links.Constants;
522     var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break, default);
523     links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
524     return setter.Result;
525 }
526 /// <param name="links">Хранилище связей.</param>
527 [MethodImpl(MethodImplOptions.AggressiveInlining)]
528 public static TLink Create<TLink>(this ILinks<TLink> links) => links.Create(null);
529 /// <param name="links">Хранилище связей.</param>
530 [MethodImpl(MethodImplOptions.AggressiveInlining)]
531 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
532 {
533     var link = links.Create();
534     return links.Update(link, link, link);
535 }
536 /// <param name="links">Хранилище связей.</param>
537 [MethodImpl(MethodImplOptions.AggressiveInlining)]
538 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
539 → target) => links.Update(links.Create(), source, target);
540 /// <summary>
541 /// Обновляет связь с указанными началом (Source) и концом (Target)
542 /// на связь с указанными началом (NewSource) и концом (NewTarget).
543 /// </summary>
544 /// <param name="links">Хранилище связей.</param>
545 /// <param name="link">Индекс обновляемой связи.</param>
546 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
547 → выполняется обновление.</param>
548 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
549 → выполняется обновление.</param>
550 /// <returns>Индекс обновлённой связи.</returns>
551 [MethodImpl(MethodImplOptions.AggressiveInlining)]
552 public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
553 → TLink newTarget) => links.Update(new LinkAddress<TLink>(link), new Link<TLink>(link,
554 → newSource, newTarget));
555 /// <summary>
556 /// Обновляет связь с указанными началом (Source) и концом (Target)
557 /// на связь с указанными началом (NewSource) и концом (NewTarget).
558 /// </summary>
559 /// <param name="links">Хранилище связей.</param>
560 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
561 → может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
562 → Itself - требование установить ссылку на себя, 1..∞ конкретный адрес другой
563 → связи.</param>
564 /// <returns>Индекс обновлённой связи.</returns>
565 [MethodImpl(MethodImplOptions.AggressiveInlining)]
566 public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
567 {
568     if (restrictions.Length == 2)
569     {
570         return links.MergeAndDelete(restrictions[0], restrictions[1]);
571     }
572     if (restrictions.Length == 4)
573     {
574         return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
575 → restrictions[2], restrictions[3]);
576     }
577     else
578     {
579         return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
580     }
581 }
582 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

577 public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
    ↳ links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
578 {
579     var equalityComparer = EqualityComparer<TLink>.Default;
580     var constants = links.Constants;
581     var restrictionsIndex = restrictions[constants.IndexPart];
582     var substitutionIndex = substitution[constants.IndexPart];
583     if (equalityComparer.Equals(substitutionIndex, default))
584     {
585         substitutionIndex = restrictionsIndex;
586     }
587     var source = substitution[constants.SourcePart];
588     var target = substitution[constants.TargetPart];
589     source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
590     target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
591     return new Link<TLink>(substitutionIndex, source, target);
592 }
593
594 /// <summary>
595 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
    ↳ с указанными Source (началом) и Target (концом).
596 /// </summary>
597 /// <param name="links">Хранилище связей.</param>
598 /// <param name="source">Индекс связи, которая является началом на создаваемой
    ↳ связи.</param>
599 /// <param name="target">Индекс связи, которая является концом для создаваемой
    ↳ связи.</param>
600 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
601 [MethodImpl(MethodImplOptions.AggressiveInlining)]
602 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target)
603 {
604     var link = links.SearchOrDefault(source, target);
605     if (EqualityComparer<TLink>.Default.Equals(link, default))
606     {
607         link = links.CreateAndUpdate(source, target);
608     }
609     return link;
610 }
611
612 /// <summary>
613 /// Обновляет связь с указанными началом (Source) и концом (Target)
614 /// на связь с указанными началом (NewSource) и концом (NewTarget).
615 /// </summary>
616 /// <param name="links">Хранилище связей.</param>
617 /// <param name="source">Индекс связи, которая является началом обновляемой
    ↳ связи.</param>
618 /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
619 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    ↳ выполняется обновление.</param>
620 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    ↳ выполняется обновление.</param>
621 /// <returns>Индекс обновлённой связи.</returns>
622 [MethodImpl(MethodImplOptions.AggressiveInlining)]
623 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
    ↳ TLink target, TLink newSource, TLink newTarget)
624 {
625     var equalityComparer = EqualityComparer<TLink>.Default;
626     var link = links.SearchOrDefault(source, target);
627     if (equalityComparer.Equals(link, default))
628     {
629         return links.CreateAndUpdate(newSource, newTarget);
630     }
631     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
    ↳ target))
632     {
633         return link;
634     }
635     return links.Update(link, newSource, newTarget);
636 }
637
638 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
639 /// <param name="links">Хранилище связей.</param>
640 /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
641 /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
642 [MethodImpl(MethodImplOptions.AggressiveInlining)]
643 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target)

```

```

644 {
645     var link = links.SearchOrDefault(source, target);
646     if (!EqualityComparer<TLink>.Default.Equals(link, default))
647     {
648         links.Delete(link);
649         return link;
650     }
651     return default;
652 }
653
654 /// <summary>Удаляет несколько связей.</summary>
655 /// <param name="links">Хранилище связей.</param>
656 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
657 [MethodImpl(MethodImplOptions.AggressiveInlining)]
658 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
659 {
660     for (int i = 0; i < deletedLinks.Count; i++)
661     {
662         links.Delete(deletedLinks[i]);
663     }
664 }
665
666 /// <remarks>Before execution of this method ensure that deleted link is detached (all
667 ↪ values - source and target are reset to null) or it might enter into infinite
668 ↪ recursion.</remarks>
669 [MethodImpl(MethodImplOptions.AggressiveInlining)]
670 public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
671 {
672     var anyConstant = links.Constants.Any;
673     var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
674     links.DeleteByQuery(usagesAsSourceQuery);
675     var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
676     links.DeleteByQuery(usagesAsTargetQuery);
677 }
678
679 [MethodImpl(MethodImplOptions.AggressiveInlining)]
680 public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
681 {
682     var count = CheckedConverter<TLink, long>.Default.Convert(links.Count(query));
683     if (count > 0)
684     {
685         var queryResult = new TLink[count];
686         var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
687             ↪ links.Constants.Continue);
688         links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
689         for (var i = count - 1; i >= 0; i--)
690         {
691             links.Delete(queryResult[i]);
692         }
693     }
694 }
695
696 // TODO: Move to Platform.Data
697 [MethodImpl(MethodImplOptions.AggressiveInlining)]
698 public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
699 {
700     var nullConstant = links.Constants.Null;
701     var equalityComparer = EqualityComparer<TLink>.Default;
702     var link = links.GetLink(linkIndex);
703     for (int i = 1; i < link.Count; i++)
704     {
705         if (!equalityComparer.Equals(link[i], nullConstant))
706         {
707             return false;
708         }
709     }
710     return true;
711 }
712
713 // TODO: Create a universal version of this method in Platform.Data (with using of for
714 ↪ loop)
715 [MethodImpl(MethodImplOptions.AggressiveInlining)]
716 public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
717 {
718     var nullConstant = links.Constants.Null;
719     var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
720     links.Update(updateRequest);
721 }

```



```

718 // TODO: Create a universal version of this method in Platform.Data (with using of for
719 ↪ loop)
720 [MethodImpl(MethodImplOptions.AggressiveInlining)]
721 public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
722 {
723     if (!links.AreValuesReset(linkIndex))
724     {
725         links.ResetValues(linkIndex);
726     }
727 }
728
729 /// <summary>
730 /// Merging two usages graphs, all children of old link moved to be children of new link
731 ↪ or deleted.
732 /// </summary>
733 [MethodImpl(MethodImplOptions.AggressiveInlining)]
734 public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
735 ↪ TLink newLinkIndex)
736 {
737     var addressToInt64Converter = CheckedConverter<TLink, long>.Default;
738     var equalityComparer = EqualityComparer<TLink>.Default;
739     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
740     {
741         var constants = links.Constants;
742         var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
743 ↪ constants.Any);
744         var usagesAsSourceCount =
745 ↪ addressToInt64Converter.Convert(links.Count(usagesAsSourceQuery));
746         var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
747 ↪ oldLinkIndex);
748         var usagesAsTargetCount =
749 ↪ addressToInt64Converter.Convert(links.Count(usagesAsTargetQuery));
750         var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
751 ↪ usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
752         if (!isStandalonePoint)
753         {
754             var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
755             if (totalUsages > 0)
756             {
757                 var usages = ArrayPool.Allocate<TLink>(totalUsages);
758                 var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
759 ↪ links.Constants.Continue);
760                 var i = 0L;
761                 if (usagesAsSourceCount > 0)
762                 {
763                     links.Each(usagesFiller.AddFirstAndReturnConstant,
764 ↪ usagesAsSourceQuery);
765                     for (; i < usagesAsSourceCount; i++)
766                     {
767                         var usage = usages[i];
768                         if (!equalityComparer.Equals(usage, oldLinkIndex))
769                         {
770                             links.Update(usage, newLinkIndex, links.GetTarget(usage));
771                         }
772                     }
773                 }
774                 if (usagesAsTargetCount > 0)
775                 {
776                     links.Each(usagesFiller.AddFirstAndReturnConstant,
777 ↪ usagesAsTargetQuery);
778                     for (; i < usages.Length; i++)
779                     {
780                         var usage = usages[i];
781                         if (!equalityComparer.Equals(usage, oldLinkIndex))
782                         {
783                             links.Update(usage, links.GetSource(usage), newLinkIndex);
784                         }
785                     }
786                 }
787             }
788             ArrayPool.Free(usages);
789         }
790     }
791     return newLinkIndex;
792 }
793
794 /// <summary>

```

```

785     /// Replace one link with another (replaced link is deleted, children are updated or
    ↪ deleted).
786     /// </summary>
787     [MethodImpl(MethodImplOptions.AggressiveInlining)]
788     public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
    ↪ TLink newLinkIndex)
789     {
790         var equalityComparer = EqualityComparer<TLink>.Default;
791         if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
792         {
793             links.MergeUsages(oldLinkIndex, newLinkIndex);
794             links.Delete(oldLinkIndex);
795         }
796         return newLinkIndex;
797     }
798
799     [MethodImpl(MethodImplOptions.AggressiveInlining)]
800     public static ILinks<TLink>
    ↪ DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
801     {
802         links = new LinksCascadeUsagesResolver<TLink>(links);
803         links = new NonNullContentsLinkDeletionResolver<TLink>(links);
804         links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
805         return links;
806     }
807
808     [MethodImpl(MethodImplOptions.AggressiveInlining)]
809     public static string Format<TLink>(this ILinks<TLink> links, IList<TLink> link)
810     {
811         var constants = links.Constants;
812         return $"({link[constants.IndexPart]}: {link[constants.SourcePart]}
    ↪ {link[constants.TargetPart]});";
813     }
814
815     [MethodImpl(MethodImplOptions.AggressiveInlining)]
816     public static string Format<TLink>(this ILinks<TLink> links, TLink link) =>
    ↪ links.Format(links.GetLink(link));
817 }
818 }

```

## 1.21 ./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  }

```

## 1.22 ./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 =
    ↪ EqualityComparer<TLink>.Default;
12
13         private readonly TLink _frequencyMarker;
14         private readonly TLink _unaryOne;
15         private readonly IIncrementer<TLink> _unaryNumberIncrementer;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
    ↪ IIncrementer<TLink> unaryNumberIncrementer)
19             : base(links)
20         {
21             _frequencyMarker = frequencyMarker;
22             _unaryOne = unaryOne;
23             _unaryNumberIncrementer = unaryNumberIncrementer;
24         }
25

```

```

26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     public TLink Increment(TLink frequency)
28     {
29         var links = _links;
30         if (_equalityComparer.Equals(frequency, default))
31         {
32             return links.GetOrCreate(_unaryOne, _frequencyMarker);
33         }
34         var incrementedSource =
35             ↪ _unaryNumberIncrementer.Increment(links.GetSource(frequency));
36         return links.GetOrCreate(incrementedSource, _frequencyMarker);
37     }
38 }

```

### 1.23 ./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             var target = links.GetTarget(unaryNumber);
30             if (_equalityComparer.Equals(source, target))
31             {
32                 return links.GetOrCreate(unaryNumber, _unaryOne);
33             }
34             else
35             {
36                 return links.GetOrCreate(source, Increment(target));
37             }
38         }
39     }
40 }

```

### 1.24 ./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;

```

```

22 private static readonly EqualityComparer<TLink> _equalityComparer =
    ↳ EqualityComparer<TLink>.Default;
23
24 private const int Length = 3;
25
26 public readonly TLink Index;
27 public readonly TLink Source;
28 public readonly TLink Target;
29
30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 public Link(params TLink[] values) => SetValues(values, out Index, out Source, out
    ↳ Target);
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 public Link(ICollection<TLink> values) => SetValues(values, out Index, out Source, out Target);
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 public Link(object other)
38 {
39     if (other is Link<TLink> otherLink)
40     {
41         SetValues(ref otherLink, out Index, out Source, out Target);
42     }
43     else if (other is ICollection<TLink> otherList)
44     {
45         SetValues(otherList, out Index, out Source, out Target);
46     }
47     else
48     {
49         throw new NotSupportedException();
50     }
51 }
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 public Link(ref Link<TLink> other) => SetValues(ref other, out Index, out Source, out
    ↳ Target);
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 public Link(TLink index, TLink source, TLink target)
58 {
59     Index = index;
60     Source = source;
61     Target = target;
62 }
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 private static void SetValues(ref Link<TLink> other, out TLink index, out TLink source,
    ↳ out TLink target)
66 {
67     index = other.Index;
68     source = other.Source;
69     target = other.Target;
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 private static void SetValues(ICollection<TLink> values, out TLink index, out TLink source,
    ↳ out TLink target)
74 {
75     switch (values.Count)
76     {
77         case 3:
78             index = values[0];
79             source = values[1];
80             target = values[2];
81             break;
82         case 2:
83             index = values[0];
84             source = values[1];
85             target = default;
86             break;
87         case 1:
88             index = values[0];
89             source = default;
90             target = default;
91             break;
92         default:
93             index = default;
94             source = default;
95             target = default;
96             break;

```

```

    }
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public override int GetHashCode() => (Index, Source, Target).GetHashCode();

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
    && _equalityComparer.Equals(Source, _constants.Null)
    && _equalityComparer.Equals(Target, _constants.Null);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public override bool Equals(object other) => other is Link<TLink> &&
    => Equals((Link<TLink>)other);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
    && _equalityComparer.Equals(Source, other.Source)
    && _equalityComparer.Equals(Target, other.Target);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static string ToString(TLink index, TLink source, TLink target) => $"({index}:
    ↳ {source}->{target})";

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static string ToString(TLink source, TLink target) => $"({source}->{target})";

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static implicit operator TLink[] (Link<TLink> link) => link.ToArray();

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static implicit operator Link<TLink>(TLink[] linkArray) => new
    ↳ Link<TLink>(linkArray);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
    ↳ ToString(Source, Target) : ToString(Index, Source, Target);

#region IList

public int Count
{
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    get => Length;
}

public bool IsReadOnly
{
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    get => true;
}

public TLink this[int index]
{
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    get
    {
        Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
            ↳ nameof(index));
        if (index == _constants.IndexPart)
        {
            return Index;
        }
        if (index == _constants.SourcePart)
        {
            return Source;
        }
        if (index == _constants.TargetPart)
        {
            return Target;
        }
        throw new NotSupportedException(); // Impossible path due to
            ↳ Ensure.ArgumentInRange
    }
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    set => throw new NotSupportedException();
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

170     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
171
172     [MethodImpl(MethodImplOptions.AggressiveInlining)]
173     public IEnumerator<TLink> GetEnumerator()
174     {
175         yield return Index;
176         yield return Source;
177         yield return Target;
178     }
179
180     [MethodImpl(MethodImplOptions.AggressiveInlining)]
181     public void Add(TLink item) => throw new NotSupportedException();
182
183     [MethodImpl(MethodImplOptions.AggressiveInlining)]
184     public void Clear() => throw new NotSupportedException();
185
186     [MethodImpl(MethodImplOptions.AggressiveInlining)]
187     public bool Contains(TLink item) => IndexOf(item) >= 0;
188
189     [MethodImpl(MethodImplOptions.AggressiveInlining)]
190     public void CopyTo(TLink[] array, int arrayIndex)
191     {
192         Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
193         Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
194             ↪ nameof(arrayIndex));
195         if (arrayIndex + Length > array.Length)
196         {
197             throw new InvalidOperationException();
198         }
199         array[arrayIndex++] = Index;
200         array[arrayIndex++] = Source;
201         array[arrayIndex] = Target;
202     }
203
204     [MethodImpl(MethodImplOptions.AggressiveInlining)]
205     public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
206
207     [MethodImpl(MethodImplOptions.AggressiveInlining)]
208     public int IndexOf(TLink item)
209     {
210         if (_equalityComparer.Equals(Index, item))
211         {
212             return _constants.IndexPart;
213         }
214         if (_equalityComparer.Equals(Source, item))
215         {
216             return _constants.SourcePart;
217         }
218         if (_equalityComparer.Equals(Target, item))
219         {
220             return _constants.TargetPart;
221         }
222         return -1;
223     }
224
225     [MethodImpl(MethodImplOptions.AggressiveInlining)]
226     public void Insert(int index, TLink item) => throw new NotSupportedException();
227
228     [MethodImpl(MethodImplOptions.AggressiveInlining)]
229     public void RemoveAt(int index) => throw new NotSupportedException();
230
231     [MethodImpl(MethodImplOptions.AggressiveInlining)]
232     public static bool operator ==(Link<TLink> left, Link<TLink> right) =>
233         ↪ left.Equals(right);
234
235     [MethodImpl(MethodImplOptions.AggressiveInlining)]
236     public static bool operator !=(Link<TLink> left, Link<TLink> right) => !(left == right);
237
238     #endregion
239 }

```

## 1.25 ./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 }

```

## 1.26 ./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 }

```

## 1.27 ./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 }

```

## 1.28 ./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 }

```

### 1.29 ./csharp/Platform.Data.Doublets/Memory/IndexTreeType.cs

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets.Memory
4  {
5      public enum IndexTreeType
6      {
7          Default = 0,
8          SizeBalancedTree = 1,
9          RecursionlessSizeBalancedTree = 2,
10         SizedAndThreadedAVLBalancedTree = 3
11     }
12 }
```

### 1.30 ./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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public static bool operator ==(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
47             ↳ left.Equals(right);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public static bool operator !=(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
51             ↳ !(left == right);
52     }
53 }
```

### 1.31 ./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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
72         protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
73         {
74             ref var firstLink = ref GetLinkDataPartReference(first);
75             ref var secondLink = ref GetLinkDataPartReference(second);
76             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
77             ↳ secondLink.Source, secondLink.Target);
78         }
79
80         [MethodImpl(MethodImplOptions.AggressiveInlining)]
81         protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
82         {
83             ref var firstLink = ref GetLinkDataPartReference(first);
84             ref var secondLink = ref GetLinkDataPartReference(second);
85             return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
86             ↳ secondLink.Source, secondLink.Target);
87         }
88
89         public TLink this[TLink index]
90     }
91 }

```

```

78 {
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     get
81     {
82         var root = GetTreeRoot();
83         if (GreaterOrEqualThan(index, GetSize(root)))
84         {
85             return Zero;
86         }
87         while (!EqualToZero(root))
88         {
89             var left = GetLeftOrDefault(root);
90             var leftSize = GetSizeOrZero(left);
91             if (LessThan(index, leftSize))
92             {
93                 root = left;
94                 continue;
95             }
96             if (AreEqual(index, leftSize))
97             {
98                 return root;
99             }
100             root = GetRightOrDefault(root);
101             index = Subtract(index, Increment(leftSize));
102         }
103         return Zero; // TODO: Impossible situation exception (only if tree structure
104             ↳ broken)
105     }
106
107     /// <summary>
108     /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
109     /// ↳ (концом).
110     /// </summary>
111     /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
112     /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
113     /// <returns>Индекс искомой связи.</returns>
114     [MethodImpl(MethodImplOptions.AggressiveInlining)]
115     public TLink Search(TLink source, TLink target)
116     {
117         var root = GetTreeRoot();
118         while (!EqualToZero(root))
119         {
120             ref var rootLink = ref GetLinkDataPartReference(root);
121             var rootSource = rootLink.Source;
122             var rootTarget = rootLink.Target;
123             if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
124                 ↳ node.Key < root.Key
125             {
126                 root = GetLeftOrDefault(root);
127             }
128             else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
129                 ↳ node.Key > root.Key
130             {
131                 root = GetRightOrDefault(root);
132             }
133             else // node.Key == root.Key
134             {
135                 return root;
136             }
137         }
138         return Zero;
139     }
140
141     /// TODO: Return indices range instead of references count
142     [MethodImpl(MethodImplOptions.AggressiveInlining)]
143     public TLink CountUsages(TLink link)
144     {
145         var root = GetTreeRoot();
146         var total = GetSize(root);
147         var totalRightIgnore = Zero;
148         while (!EqualToZero(root))
149         {
150             var @base = GetBasePartValue(root);
151             if (LessOrEqualThan(@base, link))
152             {
153                 root = GetRightOrDefault(root);
154             }
155         }
156     }

```

```

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

```

```

229         sb.Append('>');
230         sb.Append(link.Target);
231     }
232 }
233 }

```

### 1.32 ./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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         protected override bool FirstIsToLeftOfSecond(TLink firstSource, TLink firstTarget,
56             ↪ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
57             ↪ (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
61             ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
62             ↪ (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected override void ClearNode(TLink node)
66         {
67             ref var link = ref GetLinkIndexPartReference(node);
68             link.LeftAsSource = Zero;
69             link.RightAsSource = Zero;
70             link.SizeAsSource = Zero;
71         }
72     }
73 }

```

```

56     }
57 }

1.33 ./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)]
55         protected override bool FirstIsToLeftOfSecond(TLink firstSource, TLink firstTarget,
56             ↪ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
57             ↪ (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
61             ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
62             ↪ (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected override void ClearNode(TLink node)
66         {
67             ref var link = ref GetLinkIndexPartReference(node);
68             link.LeftAsTarget = Zero;
69             link.RightAsTarget = Zero;
70             link.SizeAsTarget = Zero;
71         }
72     }
73 }

```

## 1.34 ./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 FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
57             ↳ LessThan(GetKeyPartValue(first), GetKeyPartValue(second));
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
61             ↳ GreaterThan(GetKeyPartValue(first), GetKeyPartValue(second));
62
63         [MethodImpl(MethodImplOptions.AggressiveInlining)]
64         protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
65         {
66             ref var link = ref GetLinkDataPartReference(linkIndex);
67             return new Link<TLink>(linkIndex, link.Source, link.Target);
68         }
69
70         public TLink this[TLink link, TLink index]
71         {
72             [MethodImpl(MethodImplOptions.AggressiveInlining)]
73             get
74             {
75                 var root = GetTreeRoot(link);
76                 if (GreaterOrEqualThan(index, GetSize(root)))
77                 {
78                     return Zero;
79                 }
80             }
81         }
82     }
83 }

```

```

71     while (!EqualToZero(root))
72     {
73         var left = GetLeftOrDefault(root);
74         var leftSize = GetSizeOrZero(left);
75         if (LessThan(index, leftSize))
76         {
77             root = left;
78             continue;
79         }
80         if (AreEqual(index, leftSize))
81         {
82             return root;
83         }
84         root = GetRightOrDefault(root);
85         index = Subtract(index, Increment(leftSize));
86     }
87     return Zero; // TODO: Impossible situation exception (only if tree structure
88                 ↪ broken)
89 }
90
91 /// <summary>
92 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
93 ↪ (концом).
94 /// </summary>
95 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
96 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
97 /// <returns>Индекс искомой связи.</returns>
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 public abstract TLink Search(TLink source, TLink target);
100
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 protected TLink SearchCore(TLink root, TLink key)
103 {
104     while (!EqualToZero(root))
105     {
106         var rootKey = GetKeyPartValue(root);
107         if (LessThan(key, rootKey)) // node.Key < root.Key
108         {
109             root = GetLeftOrDefault(root);
110         }
111         else if (GreaterThan(key, rootKey)) // node.Key > root.Key
112         {
113             root = GetRightOrDefault(root);
114         }
115         else // node.Key == root.Key
116         {
117             return root;
118         }
119     }
120     return Zero;
121 }
122
123 // TODO: Return indices range instead of references count
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 public TLink CountUsages(TLink link) => GetSizeOrZero(GetTreeRoot(link));
126
127 [MethodImpl(MethodImplOptions.AggressiveInlining)]
128 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
129     ↪ EachUsageCore(@base, GetTreeRoot(@base), handler);
130
131 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
132 ↪ low-level MSIL stack.
133 [MethodImpl(MethodImplOptions.AggressiveInlining)]
134 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
135 {
136     var @continue = Continue;
137     if (EqualToZero(link))
138     {
139         return @continue;
140     }
141     var @break = Break;
142     if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
143     {
144         return @break;
145     }
146     if (AreEqual(handler(GetLinkValues(link)), @break))
147     {
148         return @break;
149     }
150 }

```

```

146     }
147     if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
148     {
149         return @break;
150     }
151     return @continue;
152 }
153
154 [MethodImpl(MethodImplOptions.AggressiveInlining)]
155 protected override void PrintNodeValue(TLink node, StringBuilder sb)
156 {
157     ref var link = ref GetLinkDataPartReference(node);
158     sb.Append(' ');
159     sb.Append(link.Source);
160     sb.Append('-');
161     sb.Append('>');
162     sb.Append(link.Target);
163 }
164 }
165 }

```

### 1.35 ./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)]
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(TLink link) =>
49             ↪ GetLinkIndexPartReference(link).RootAsSource;
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override TLink GetBasePartValue(TLink link) =>
53             ↪ GetLinkDataPartReference(link).Source;
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

43     protected override TLink GetKeyPartValue(TLink link) =>
44         ↪ GetLinkDataPartReference(link).Target;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override void ClearNode(TLink node)
48     {
49         ref var link = ref GetLinkIndexPartReference(node);
50         link.LeftAsSource = Zero;
51         link.RightAsSource = Zero;
52         link.SizeAsSource = Zero;
53     }
54
55     public override TLink Search(TLink source, TLink target) =>
56         ↪ SearchCore(GetTreeRoot(source), target);
57 }

```

### 1.36 ./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)]
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(TLink link) =>
49             ↪ GetLinkIndexPartReference(link).RootAsTarget;
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override TLink GetBasePartValue(TLink link) =>
53             ↪ GetLinkDataPartReference(link).Target;
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override TLink GetKeyPartValue(TLink link) =>
57             ↪ GetLinkDataPartReference(link).Source;
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override void ClearNode(TLink node)

```

```

47     {
48         ref var link = ref GetLinkIndexPartReference(node);
49         link.LeftAsTarget = Zero;
50         link.RightAsTarget = Zero;
51         link.SizeAsTarget = Zero;
52     }
53
54     public override TLink Search(TLink source, TLink target) =>
55         ↪ SearchCore(GetTreeRoot(target), source);
56 }

```

### 1.37 ./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             _createExternalTargetTreeMethods = () => new
45                 ↪ ExternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
46                 ↪ _linksIndexParts, _header);
47             Init(dataMemory, indexMemory);
48         }
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override void SetPointers(IResizableDirectMemory dataMemory,
52             ↪ IResizableDirectMemory indexMemory)
53         {
54             _linksDataParts = (byte*)dataMemory.Pointer;
55             _linksIndexParts = (byte*)indexMemory.Pointer;
56             _header = _linksIndexParts;
57             InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
58             ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
59             InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
60             ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
61             UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_linksDataParts, _header);
62         }
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected override void ResetPointers()
66         {
67
68         }
69     }
70 }

```

```

53     base.ResetPointers();
54     _linksDataParts = null;
55     _linksIndexParts = null;
56     _header = null;
57 }
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected override ref LinksHeader<TLink> GetHeaderReference() => ref
    ↳ AsRef<LinksHeader<TLink>>(_header);
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
    ↳ => ref AsRef<RawLinkDataPart<TLink>>(_linksDataParts + (LinkDataPartSizeInBytes *
    ↳ ConvertToInt64(linkIndex)));
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
    ↳ linkIndex) => ref AsRef<RawLinkIndexPart<TLink>>(_linksIndexParts +
    ↳ (LinkIndexPartSizeInBytes * ConvertToInt64(linkIndex)));
67 }
68 }

```

### 1.38 ./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;
34
35         public static readonly long LinkIndexPartSizeInBytes =
36             ↳ RawLinkIndexPart<TLink>.SizeInBytes;
37
38         public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
39
40         public static readonly long DefaultLinksSizeStep = 1 * 1024 * 1024;
41
42         protected readonly IResizableDirectMemory _dataMemory;
43         protected readonly IResizableDirectMemory _indexMemory;
44         protected readonly long _dataMemoryReservationStepInBytes;
45         protected readonly long _indexMemoryReservationStepInBytes;
46
47         protected ILinksTreeMethods<TLink> InternalSourcesTreeMethods;
48         protected ILinksTreeMethods<TLink> ExternalSourcesTreeMethods;
49         protected ILinksTreeMethods<TLink> InternalTargetsTreeMethods;
50         protected ILinksTreeMethods<TLink> ExternalTargetsTreeMethods;
51         // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
52         ↳ нужно использовать не список а дерево, так как так можно быстрее проверить на
53         ↳ наличие связи внутри
54         protected ILinksListMethods<TLink> UnusedLinksListMethods;
55
56         /// <summary>
57         /// Возвращает общее число связей находящихся в хранилище.
58         /// </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
71     ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants)
72     {
73         _dataMemory = dataMemory;
74         _indexMemory = indexMemory;
75         _dataMemoryReservationStepInBytes = memoryReservationStep * LinkDataPartSizeInBytes;
76         _indexMemoryReservationStepInBytes = memoryReservationStep *
77         ↪ LinkIndexPartSizeInBytes;
78         Constants = constants;
79     }
80
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
83     ↪ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
84     ↪ memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }
85
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     protected virtual void Init(IResizableDirectMemory dataMemory, IResizableDirectMemory
88     ↪ indexMemory)
89     {
90         // Read allocated links from header
91         if (indexMemory.ReservedCapacity < LinkHeaderSizeInBytes)
92         {
93             indexMemory.ReservedCapacity = LinkHeaderSizeInBytes;
94         }
95         SetPointers(dataMemory, indexMemory);
96         ref var header = ref GetHeaderReference();
97         var allocatedLinks = ConvertToInt64(header.AllocatedLinks);
98         // Adjust reserved capacity
99         var minimumDataReservedCapacity = allocatedLinks * LinkDataPartSizeInBytes;
100         if (minimumDataReservedCapacity < _dataMemoryReservationStepInBytes)
101         {
102             minimumDataReservedCapacity = _dataMemoryReservationStepInBytes;
103         }
104         var minimumIndexReservedCapacity = allocatedLinks * LinkDataPartSizeInBytes;
105         if (minimumIndexReservedCapacity < _indexMemoryReservationStepInBytes)
106         {
107             minimumIndexReservedCapacity = _indexMemoryReservationStepInBytes;
108         }
109         // Check for alignment
110         if (minimumDataReservedCapacity % _dataMemoryReservationStepInBytes > 0)
111         {
112             minimumDataReservedCapacity = ((minimumDataReservedCapacity /
113             ↪ _dataMemoryReservationStepInBytes) * _dataMemoryReservationStepInBytes) +
114             ↪ _dataMemoryReservationStepInBytes;
115         }
116         if (minimumIndexReservedCapacity % _indexMemoryReservationStepInBytes > 0)
117         {
118             minimumIndexReservedCapacity = ((minimumIndexReservedCapacity /
119             ↪ _indexMemoryReservationStepInBytes) * _indexMemoryReservationStepInBytes) +
120             ↪ _indexMemoryReservationStepInBytes;
121         }
122         if (dataMemory.ReservedCapacity != minimumDataReservedCapacity)
123         {
124             dataMemory.ReservedCapacity = minimumDataReservedCapacity;
125         }
126         if (indexMemory.ReservedCapacity != minimumIndexReservedCapacity)
127         {
128             indexMemory.ReservedCapacity = minimumIndexReservedCapacity;
129         }
130         SetPointers(dataMemory, indexMemory);
131         header = ref GetHeaderReference();

```

```

123 // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
124 // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
125 dataMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
    ↳ LinkDataPartSizeInBytes) + LinkDataPartSizeInBytes; // First link is read only
    ↳ zero link.
126 indexMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
    ↳ LinkIndexPartSizeInBytes) + LinkHeaderSizeInBytes;
127 // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
128 // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
129 header.ReservedLinks = ConvertToAddress((dataMemory.ReservedCapacity -
    ↳ LinkDataPartSizeInBytes) / LinkDataPartSizeInBytes);
130 }
131
132 [MethodImpl(MethodImplOptions.AggressiveInlining)]
133 public virtual TLink Count(IList<TLink> restrictions)
134 {
135     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
136     if (restrictions.Count == 0)
137     {
138         return Total;
139     }
140     var constants = Constants;
141     var any = constants.Any;
142     var index = restrictions[constants.IndexPart];
143     if (restrictions.Count == 1)
144     {
145         if (AreEqual(index, any))
146         {
147             return Total;
148         }
149         return Exists(index) ? GetOne() : GetZero();
150     }
151     if (restrictions.Count == 2)
152     {
153         var value = restrictions[1];
154         if (AreEqual(index, any))
155         {
156             if (AreEqual(value, any))
157             {
158                 return Total; // Any - как отсутствие ограничения
159             }
160             var externalReferencesRange = constants.ExternalReferencesRange;
161             if (externalReferencesRange.HasValue &&
    ↳ externalReferencesRange.Value.Contains(value))
162             {
163                 return Add(ExternalSourcesTreeMethods.CountUsages(value),
    ↳ ExternalTargetsTreeMethods.CountUsages(value));
164             }
165             else
166             {
167                 return Add(InternalSourcesTreeMethods.CountUsages(value),
    ↳ InternalTargetsTreeMethods.CountUsages(value));
168             }
169         }
170         else
171         {
172             if (!Exists(index))
173             {
174                 return GetZero();
175             }
176             if (AreEqual(value, any))
177             {
178                 return GetOne();
179             }
180             ref var storedLinkValue = ref GetLinkDataPartReference(index);
181             if (AreEqual(storedLinkValue.Source, value) ||
    ↳ AreEqual(storedLinkValue.Target, value))
182             {
183                 return GetOne();
184             }
185             return GetZero();
186         }
187     }
188     if (restrictions.Count == 3)
189     {
190         var externalReferencesRange = constants.ExternalReferencesRange;
191         var source = restrictions[constants.SourcePart];
192         var target = restrictions[constants.TargetPart];

```

```

193 if (AreEqual(index, any))
194 {
195     if (AreEqual(source, any) && AreEqual(target, any))
196     {
197         return Total;
198     }
199     else if (AreEqual(source, any))
200     {
201         if (externalReferencesRange.HasValue &&
202             ↪ externalReferencesRange.Value.Contains(target))
203         {
204             return ExternalTargetsTreeMethods.CountUsages(target);
205         }
206         else
207         {
208             return InternalTargetsTreeMethods.CountUsages(target);
209         }
210     }
211     else if (AreEqual(target, any))
212     {
213         if (externalReferencesRange.HasValue &&
214             ↪ externalReferencesRange.Value.Contains(source))
215         {
216             return ExternalSourcesTreeMethods.CountUsages(source);
217         }
218         else
219         {
220             return InternalSourcesTreeMethods.CountUsages(source);
221         }
222     }
223     else //if(source != Any && target != Any)
224     {
225         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
226         TLink link;
227         if (externalReferencesRange.HasValue)
228         {
229             if (externalReferencesRange.Value.Contains(source) &&
230                 ↪ externalReferencesRange.Value.Contains(target))
231             {
232                 link = ExternalSourcesTreeMethods.Search(source, target);
233             }
234             else if (externalReferencesRange.Value.Contains(source))
235             {
236                 link = InternalTargetsTreeMethods.Search(source, target);
237             }
238             else if (externalReferencesRange.Value.Contains(target))
239             {
240                 link = InternalSourcesTreeMethods.Search(source, target);
241             }
242             else
243             {
244                 if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
245                     ↪ InternalTargetsTreeMethods.CountUsages(target)))
246                 {
247                     link = InternalTargetsTreeMethods.Search(source, target);
248                 }
249                 else
250                 {
251                     link = InternalSourcesTreeMethods.Search(source, target);
252                 }
253             }
254         }
255         else
256         {
257             if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
258                 ↪ InternalTargetsTreeMethods.CountUsages(target)))
259             {
260                 link = InternalTargetsTreeMethods.Search(source, target);
261             }
262             else
263             {
264                 link = InternalSourcesTreeMethods.Search(source, target);
265             }
266         }
267         return AreEqual(link, constants.Null) ? GetZero() : GetOne();
268     }
269 }
270 else

```

```

266 {
267     if (!Exists(index))
268     {
269         return GetZero();
270     }
271     if (AreEqual(source, any) && AreEqual(target, any))
272     {
273         return GetOne();
274     }
275     ref var storedLinkValue = ref GetLinkDataPartReference(index);
276     if (!AreEqual(source, any) && !AreEqual(target, any))
277     {
278         if (AreEqual(storedLinkValue.Source, source) &&
279             ↪ AreEqual(storedLinkValue.Target, target))
280         {
281             return GetOne();
282         }
283         return GetZero();
284     }
285     var value = default(TLink);
286     if (AreEqual(source, any))
287     {
288         value = target;
289     }
290     if (AreEqual(target, any))
291     {
292         value = source;
293     }
294     if (AreEqual(storedLinkValue.Source, value) ||
295         ↪ AreEqual(storedLinkValue.Target, value))
296     {
297         return GetOne();
298     }
299     return GetZero();
300 }
301 }
302
303 [MethodImpl(MethodImplOptions.AggressiveInlining)]
304 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
305 {
306     var constants = Constants;
307     var @break = constants.Break;
308     if (restrictions.Count == 0)
309     {
310         for (var link = GetOne(); LessOrEqualThan(link,
311             ↪ GetHeaderReference().AllocatedLinks); link = Increment(link))
312         {
313             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
314             {
315                 return @break;
316             }
317         }
318         return @break;
319     }
320     var @continue = constants.Continue;
321     var any = constants.Any;
322     var index = restrictions[constants.IndexPart];
323     if (restrictions.Count == 1)
324     {
325         if (AreEqual(index, any))
326         {
327             return Each(handler, Array.Empty<TLink>());
328         }
329         if (!Exists(index))
330         {
331             return @continue;
332         }
333         return handler(GetLinkStruct(index));
334     }
335     if (restrictions.Count == 2)
336     {
337         var value = restrictions[1];
338         if (AreEqual(index, any))
339         {
340             if (AreEqual(value, any))

```

```

340     {
341         return Each(handler, Array.Empty<TLink>());
342     }
343     if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
344     {
345         return @break;
346     }
347     return Each(handler, new Link<TLink>(index, any, value));
348 }
349 else
350 {
351     if (!Exists(index))
352     {
353         return @continue;
354     }
355     if (AreEqual(value, any))
356     {
357         return handler(GetLinkStruct(index));
358     }
359     ref var storedLinkValue = ref GetLinkDataPartReference(index);
360     if (AreEqual(storedLinkValue.Source, value) ||
361         AreEqual(storedLinkValue.Target, value))
362     {
363         return handler(GetLinkStruct(index));
364     }
365     return @continue;
366 }
367 }
368 if (restrictions.Count == 3)
369 {
370     var externalReferencesRange = constants.ExternalReferencesRange;
371     var source = restrictions[constants.SourcePart];
372     var target = restrictions[constants.TargetPart];
373     if (AreEqual(index, any))
374     {
375         if (AreEqual(source, any) && AreEqual(target, any))
376         {
377             return Each(handler, Array.Empty<TLink>());
378         }
379         else if (AreEqual(source, any))
380         {
381             if (externalReferencesRange.HasValue &&
382                 ↪ externalReferencesRange.Value.Contains(target))
383             {
384                 return ExternalTargetsTreeMethods.EachUsage(target, handler);
385             }
386             else
387             {
388                 return InternalTargetsTreeMethods.EachUsage(target, handler);
389             }
390         }
391         else if (AreEqual(target, any))
392         {
393             if (externalReferencesRange.HasValue &&
394                 ↪ externalReferencesRange.Value.Contains(source))
395             {
396                 return ExternalSourcesTreeMethods.EachUsage(source, handler);
397             }
398             else
399             {
400                 return InternalSourcesTreeMethods.EachUsage(source, handler);
401             }
402         }
403         else //if(source != Any && target != Any)
404         {
405             TLink link;
406             if (externalReferencesRange.HasValue)
407             {
408                 if (externalReferencesRange.Value.Contains(source) &&
409                     ↪ externalReferencesRange.Value.Contains(target))
410                 {
411                     link = ExternalSourcesTreeMethods.Search(source, target);
412                 }
413                 else if (externalReferencesRange.Value.Contains(source))
414                 {
415                     link = InternalTargetsTreeMethods.Search(source, target);
416                 }
417                 else if (externalReferencesRange.Value.Contains(target))

```



```

415         {
416             link = InternalSourcesTreeMethods.Search(source, target);
417         }
418         else
419         {
420             if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
421                 ↪ InternalTargetsTreeMethods.CountUsages(target)))
422             {
423                 link = InternalTargetsTreeMethods.Search(source, target);
424             }
425             else
426             {
427                 link = InternalSourcesTreeMethods.Search(source, target);
428             }
429         }
430     }
431     else
432     {
433         if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
434             ↪ InternalTargetsTreeMethods.CountUsages(target)))
435         {
436             link = InternalTargetsTreeMethods.Search(source, target);
437         }
438         else
439         {
440             link = InternalSourcesTreeMethods.Search(source, target);
441         }
442     }
443     return AreEqual(link, constants.Null) ? @continue :
444         ↪ handler(GetLinkStruct(link));
445 }
446 else
447 {
448     if (!Exists(index))
449     {
450         return @continue;
451     }
452     if (AreEqual(source, any) && AreEqual(target, any))
453     {
454         return handler(GetLinkStruct(index));
455     }
456     ref var storedLinkValue = ref GetLinkDataPartReference(index);
457     if (!AreEqual(source, any) && !AreEqual(target, any))
458     {
459         if (AreEqual(storedLinkValue.Source, source) &&
460             AreEqual(storedLinkValue.Target, target))
461         {
462             return handler(GetLinkStruct(index));
463         }
464         return @continue;
465     }
466     var value = default(TLink);
467     if (AreEqual(source, any))
468     {
469         value = target;
470     }
471     if (AreEqual(target, any))
472     {
473         value = source;
474     }
475     if (AreEqual(storedLinkValue.Source, value) ||
476         AreEqual(storedLinkValue.Target, value))
477     {
478         return handler(GetLinkStruct(index));
479     }
480     return @continue;
481 }
482 }
483
484 /// <remarks>
485 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
486 ↪ в другом месте (но не в менеджере памяти, а в логике Links)
487 /// </remarks>
488 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

488 public virtual TLink Update(ICollection<TLink> restrictions, ICollection<TLink> substitution)
489 {
490     var constants = Constants;
491     var @null = constants.Null;
492     var externalReferencesRange = constants.ExternalReferencesRange;
493     var linkIndex = restrictions[constants.IndexPart];
494     ref var link = ref GetLinkDataPartReference(linkIndex);
495     var source = link.Source;
496     var target = link.Target;
497     ref var header = ref GetHeaderReference();
498     ref var rootAsSource = ref header.RootAsSource;
499     ref var rootAsTarget = ref header.RootAsTarget;
500     // Будет корректно работать только в том случае, если пространство выделенной связи
501     ↪ предварительно заполнено нулями
502     if (!AreEqual(source, @null))
503     {
504         if (externalReferencesRange.HasValue &&
505             ↪ externalReferencesRange.Value.Contains(source))
506         {
507             ExternalSourcesTreeMethods.Detach(ref rootAsSource, linkIndex);
508         }
509         else
510         {
511             InternalSourcesTreeMethods.Detach(ref
512                 ↪ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
513         }
514     }
515     if (!AreEqual(target, @null))
516     {
517         if (externalReferencesRange.HasValue &&
518             ↪ externalReferencesRange.Value.Contains(target))
519         {
520             ExternalTargetsTreeMethods.Detach(ref rootAsTarget, linkIndex);
521         }
522         else
523         {
524             InternalTargetsTreeMethods.Detach(ref
525                 ↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
526         }
527     }
528     source = link.Source = substitution[constants.SourcePart];
529     target = link.Target = substitution[constants.TargetPart];
530     if (!AreEqual(source, @null))
531     {
532         if (externalReferencesRange.HasValue &&
533             ↪ externalReferencesRange.Value.Contains(source))
534         {
535             ExternalSourcesTreeMethods.Attach(ref rootAsSource, linkIndex);
536         }
537         else
538         {
539             InternalSourcesTreeMethods.Attach(ref
540                 ↪ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
541         }
542     }
543     if (!AreEqual(target, @null))
544     {
545         if (externalReferencesRange.HasValue &&
546             ↪ externalReferencesRange.Value.Contains(target))
547         {
548             ExternalTargetsTreeMethods.Attach(ref rootAsTarget, linkIndex);
549         }
550         else
551         {
552             InternalTargetsTreeMethods.Attach(ref
553                 ↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
554         }
555     }
556     return linkIndex;
557 }
558
559 /// <remarks>
560 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
561 ↪ пространство
562 /// </remarks>
563 [MethodImpl(MethodImplOptions.AggressiveInlining)]
564 public virtual TLink Create(ICollection<TLink> restrictions)
565 {

```

```

556     ref var header = ref GetHeaderReference();
557     var freeLink = header.FirstFreeLink;
558     if (!AreEqual(freeLink, Constants.Null))
559     {
560         UnusedLinksListMethods.Detach(freeLink);
561     }
562     else
563     {
564         var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
565         if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
566         {
567             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
568         }
569         if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
570         {
571             _dataMemory.ReservedCapacity += _dataMemory.ReservationStepInBytes;
572             _indexMemory.ReservedCapacity += _indexMemory.ReservationStepInBytes;
573             SetPointers(_dataMemory, _indexMemory);
574             header = ref GetHeaderReference();
575             header.ReservedLinks = ConvertToAddress(_dataMemory.ReservedCapacity /
576                 ↪ LinkDataPartSizeInBytes);
577         }
578         header.AllocatedLinks = Increment(header.AllocatedLinks);
579         _dataMemory.UsedCapacity += LinkDataPartSizeInBytes;
580         _indexMemory.UsedCapacity += LinkIndexPartSizeInBytes;
581         freeLink = header.AllocatedLinks;
582     }
583     return freeLink;
584 }
585 [MethodImpl(MethodImplOptions.AggressiveInlining)]
586 public virtual void Delete(IList<TLink> restrictions)
587 {
588     ref var header = ref GetHeaderReference();
589     var link = restrictions[Constants.IndexPart];
590     if (LessThan(link, header.AllocatedLinks)
591     {
592         UnusedLinksListMethods.AttachAsFirst(link);
593     }
594     else if (AreEqual(link, header.AllocatedLinks))
595     {
596         header.AllocatedLinks = Decrement(header.AllocatedLinks);
597         _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
598         _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
599         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
600         ↪ пока не дойдём до первой существующей связи
601         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
602         while (GreaterThan(header.AllocatedLinks, GetZero()) &&
603             ↪ IsUnusedLink(header.AllocatedLinks))
604         {
605             UnusedLinksListMethods.Detach(header.AllocatedLinks);
606             header.AllocatedLinks = Decrement(header.AllocatedLinks);
607             _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
608             _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
609         }
610     }
611 }
612 [MethodImpl(MethodImplOptions.AggressiveInlining)]
613 public IList<TLink> GetLinkStruct(TLink linkIndex)
614 {
615     ref var link = ref GetLinkDataPartReference(linkIndex);
616     return new Link<TLink>(linkIndex, link.Source, link.Target);
617 }
618 /// <remarks>
619 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
620 ↪ адрес реально поменялся
621 ///
622 /// Указатель this.links может быть в том же месте,
623 /// так как 0-я связь не используется и имеет такой же размер как Header,
624 /// поэтому header размещается в том же месте, что и 0-я связь
625 /// </remarks>
626 [MethodImpl(MethodImplOptions.AggressiveInlining)]
627 protected abstract void SetPointers(IResizableDirectMemory dataMemory,
628     ↪ IResizableDirectMemory indexMemory);
629 [MethodImpl(MethodImplOptions.AggressiveInlining)]
630 protected virtual void ResetPointers()

```

```

630 {
631     InternalSourcesTreeMethods = null;
632     ExternalSourcesTreeMethods = null;
633     InternalTargetsTreeMethods = null;
634     ExternalTargetsTreeMethods = null;
635     UnusedLinksListMethods = null;
636 }
637
638 [MethodImpl(MethodImplOptions.AggressiveInlining)]
639 protected abstract ref LinksHeader<TLink> GetHeaderReference();
640
641 [MethodImpl(MethodImplOptions.AggressiveInlining)]
642 protected abstract ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex);
643
644 [MethodImpl(MethodImplOptions.AggressiveInlining)]
645 protected abstract ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
    ↪ linkIndex);
646
647 [MethodImpl(MethodImplOptions.AggressiveInlining)]
648 protected virtual bool Exists(TLink link)
649     => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
650     && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
651     && !IsUnusedLink(link);
652
653 [MethodImpl(MethodImplOptions.AggressiveInlining)]
654 protected virtual bool IsUnusedLink(TLink linkIndex)
655 {
656     if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
        ↪ is not needed
657     {
658         // TODO: Reduce access to memory in different location (should be enough to use
        ↪ just linkIndexPart)
659         ref var linkDataPart = ref GetLinkDataPartReference(linkIndex);
660         ref var linkIndexPart = ref GetLinkIndexPartReference(linkIndex);
661         return AreEqual(linkIndexPart.SizeAsSource, default) &&
        ↪ !AreEqual(linkDataPart.Source, default);
662     }
663     else
664     {
665         return true;
666     }
667 }
668
669 [MethodImpl(MethodImplOptions.AggressiveInlining)]
670 protected virtual TLink GetOne() => _one;
671
672 [MethodImpl(MethodImplOptions.AggressiveInlining)]
673 protected virtual TLink GetZero() => default;
674
675 [MethodImpl(MethodImplOptions.AggressiveInlining)]
676 protected virtual bool AreEqual(TLink first, TLink second) =>
    ↪ _equalityComparer.Equals(first, second);
677
678 [MethodImpl(MethodImplOptions.AggressiveInlining)]
679 protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
    ↪ second) < 0;
680
681 [MethodImpl(MethodImplOptions.AggressiveInlining)]
682 protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
    ↪ _comparer.Compare(first, second) <= 0;
683
684 [MethodImpl(MethodImplOptions.AggressiveInlining)]
685 protected virtual bool GreaterThan(TLink first, TLink second) =>
    ↪ _comparer.Compare(first, second) > 0;
686
687 [MethodImpl(MethodImplOptions.AggressiveInlining)]
688 protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
    ↪ _comparer.Compare(first, second) >= 0;
689
690 [MethodImpl(MethodImplOptions.AggressiveInlining)]
691 protected virtual long ConvertToInt64(TLink value) =>
    ↪ _addressToInt64Converter.Convert(value);
692
693 [MethodImpl(MethodImplOptions.AggressiveInlining)]
694 protected virtual TLink ConvertToAddress(long value) =>
    ↪ _int64ToAddressConverter.Convert(value);
695
696 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

697     protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
698         ↪ second);
699
700     [MethodImpl(MethodImplOptions.AggressiveInlining)]
701     protected virtual TLink Subtract(TLink first, TLink second) =>
702         ↪ Arithmetic<TLink>.Subtract(first, second);
703
704     [MethodImpl(MethodImplOptions.AggressiveInlining)]
705     protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
706
707     [MethodImpl(MethodImplOptions.AggressiveInlining)]
708     protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
709
710     #region Disposable
711     protected override bool AllowMultipleDisposeCalls
712     {
713         [MethodImpl(MethodImplOptions.AggressiveInlining)]
714         get => true;
715     }
716
717     [MethodImpl(MethodImplOptions.AggressiveInlining)]
718     protected override void Dispose(bool manual, bool wasDisposed)
719     {
720         if (!wasDisposed)
721         {
722             ResetPointers();
723             _dataMemory.DisposeIfPossible();
724             _indexMemory.DisposeIfPossible();
725         }
726     }
727     #endregion
728 }
729 }

```

### 1.39 ./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;

```

```

38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override TLink GetNext(TLink element) =>
40         ↪ GetLinkDataPartReference(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         ↪ GetLinkDataPartReference(element).Source = previous;
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override void SetNext(TLink element, TLink next) =>
59         ↪ GetLinkDataPartReference(element).Target = next;
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
63 }

```

#### 1.40 ./csharp/Platform.Data.Doublets/Memory/Split/RawLinkDataPart.cs

```

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

```

#### 1.41 ./csharp/Platform.Data.Doublets/Memory/Split/RawLinkIndexPart.cs

```

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

```

```

8 namespace Platform.Data.Doublets.Memory.Split
9 {
10     public struct RawLinkIndexPart<TLink> : IEquatable<RawLinkIndexPart<TLink>>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↳ EqualityComparer<TLink>.Default;
14
15         public static readonly long SizeInBytes = Structure<RawLinkIndexPart<TLink>>.Size;
16
17         public TLink RootAsSource;
18         public TLink LeftAsSource;
19         public TLink RightAsSource;
20         public TLink SizeAsSource;
21         public TLink RootAsTarget;
22         public TLink LeftAsTarget;
23         public TLink RightAsTarget;
24         public TLink SizeAsTarget;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public override bool Equals(object obj) => obj is RawLinkIndexPart<TLink> link ?
28             ↳ Equals(link) : false;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public bool Equals(RawLinkIndexPart<TLink> other)
32             => _equalityComparer.Equals(RootAsSource, other.RootAsSource)
33             && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
34             && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
35             && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
36             && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
37             && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
38             && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
39             && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public override int GetHashCode() => (RootAsSource, LeftAsSource, RightAsSource,
43             ↳ SizeAsSource, RootAsTarget, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public static bool operator ==(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
47             ↳ right) => left.Equals(right);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public static bool operator !=(RawLinkIndexPart<TLink> left, RawLinkIndexPart<TLink>
51             ↳ right) => !(left == right);
52     }
53 }

```

#### 1.42 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSizeBalancedTreeMethodsBase

```

1 using System.Runtime.CompilerServices;
2 using Platform.Data.Doublets.Memory.Split.Generic;
3 using TLink = System.UInt32;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Memory.Split.Specific
8 {
9     public unsafe abstract class UInt32ExternalLinksSizeBalancedTreeMethodsBase :
10         ↳ ExternalLinksSizeBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
11     {
12         protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
13         protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
14         protected new readonly LinksHeader<TLink>* Header;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected UInt32ExternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink>
18             ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
19             ↳ linksIndexParts, LinksHeader<TLink>* header)
20             : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
21         {
22             LinksDataParts = linksDataParts;
23             LinksIndexParts = linksIndexParts;
24             Header = header;
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override TLink GetZero() => 0U;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override bool EqualToZero(TLink value) => value == 0U;
32     }
33 }

```

```

30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 protected override bool AreEqual(TLink first, TLink second) => first == second;
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 protected override bool GreaterThanZero(TLink value) => value > 0U;
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 protected override bool GreaterThan(TLink first, TLink second) => first > second;
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 protected override bool GreaterOrEqualThan(TLink first, TLink second) => first >= second;
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected override bool GreaterOrEqualThanZero(TLink value) => true; // value >= 0 is
44   ↳ always true for ulong
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 protected override bool LessOrEqualThanZero(TLink value) => value == 0UL; // value is
48   ↳ always >= 0 for ulong
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected override bool LessThanZero(TLink value) => false; // value < 0 is always false
55   ↳ for ulong
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 protected override bool LessThan(TLink first, TLink second) => first < second;
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected override TLink Increment(TLink value) => ++value;
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected override TLink Decrement(TLink value) => --value;
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 protected override TLink Add(TLink first, TLink second) => first + second;
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected override TLink Subtract(TLink first, TLink second) => first - second;
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 protected override ref LinksHeader<TLink> GetHeaderReference() => ref *Header;
74
75 [MethodImpl(MethodImplOptions.AggressiveInlining)]
76 protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
77   ↳ ref LinksDataParts[link];
78
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
81   ↳ ref LinksIndexParts[link];
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
85 {
86     ref var firstLink = ref LinksDataParts[first];
87     ref var secondLink = ref LinksDataParts[second];
88     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
89   ↳ secondLink.Source, secondLink.Target);
90 }
91
92 [MethodImpl(MethodImplOptions.AggressiveInlining)]
93 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
94 {
95     ref var firstLink = ref LinksDataParts[first];
96     ref var secondLink = ref LinksDataParts[second];
97     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
98   ↳ secondLink.Source, secondLink.Target);
99 }
100 }

```

#### 1.43 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSourcesSizeBalancedTreeMeth

```

1 using System.Runtime.CompilerServices;
2 using TLink = System.UInt32;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5

```



```

6 namespace Platform.Data.Doublets.Memory.Split.Specific
7 {
8     public unsafe class UInt32ExternalLinksSourcesSizeBalancedTreeMethods :
9         ↳ UInt32ExternalLinksSizeBalancedTreeMethodsBase
10    {
11        [MethodImpl(MethodImplOptions.AggressiveInlining)]
12        public UInt32ExternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
13            ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
14            ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
15            ↳ linksIndexParts, header) { }
16
17        [MethodImpl(MethodImplOptions.AggressiveInlining)]
18        protected override ref TLink GetLeftReference(TLink node) => ref
19            ↳ LinksIndexParts[node].LeftAsSource;
20
21        [MethodImpl(MethodImplOptions.AggressiveInlining)]
22        protected override ref TLink GetRightReference(TLink node) => ref
23            ↳ LinksIndexParts[node].RightAsSource;
24
25        [MethodImpl(MethodImplOptions.AggressiveInlining)]
26        protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
27
28        [MethodImpl(MethodImplOptions.AggressiveInlining)]
29        protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
30
31        [MethodImpl(MethodImplOptions.AggressiveInlining)]
32        protected override void SetLeft(TLink node, TLink left) =>
33            ↳ LinksIndexParts[node].LeftAsSource = left;
34
35        [MethodImpl(MethodImplOptions.AggressiveInlining)]
36        protected override void SetRight(TLink node, TLink right) =>
37            ↳ LinksIndexParts[node].RightAsSource = right;
38
39        [MethodImpl(MethodImplOptions.AggressiveInlining)]
40        protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
41
42        [MethodImpl(MethodImplOptions.AggressiveInlining)]
43        protected override void SetSize(TLink node, TLink size) =>
44            ↳ LinksIndexParts[node].SizeAsSource = size;
45
46        [MethodImpl(MethodImplOptions.AggressiveInlining)]
47        protected override TLink GetTreeRoot() => Header->RootAsSource;
48
49        [MethodImpl(MethodImplOptions.AggressiveInlining)]
50        protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
51
52        [MethodImpl(MethodImplOptions.AggressiveInlining)]
53        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
54            ↳ TLink secondSource, TLink secondTarget)
55            => firstSource < secondSource || firstSource == secondSource && firstTarget <
56            ↳ secondTarget;
57
58        [MethodImpl(MethodImplOptions.AggressiveInlining)]
59        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
60            ↳ TLink secondSource, TLink secondTarget)
61            => firstSource > secondSource || firstSource == secondSource && firstTarget >
62            ↳ secondTarget;
63
64        [MethodImpl(MethodImplOptions.AggressiveInlining)]
65        protected override void ClearNode(TLink node)
66        {
67            ref var link = ref LinksIndexParts[node];
68            link.LeftAsSource = Zero;
69            link.RightAsSource = Zero;
70            link.SizeAsSource = Zero;
71        }
72    }
73 }

```

#### 1.44 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksTargetsSizeBalancedTreeMethods

```

1 using System.Runtime.CompilerServices;
2 using TLink = System.UInt32;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Memory.Split.Specific
7 {
8     public unsafe class UInt32ExternalLinksTargetsSizeBalancedTreeMethods :
9         ↳ UInt32ExternalLinksSizeBalancedTreeMethodsBase

```

```

9 {
10     [MethodImpl(MethodImplOptions.AggressiveInlining)]
11     public UInt32ExternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink>
        ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
        ↳ linksIndexParts, header) { }
12
13     [MethodImpl(MethodImplOptions.AggressiveInlining)]
14     protected override ref TLink GetLeftReference(TLink node) => ref
        ↳ LinksIndexParts[node].LeftAsTarget;
15
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     protected override ref TLink GetRightReference(TLink node) => ref
        ↳ LinksIndexParts[node].RightAsTarget;
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsTarget;
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsTarget;
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     protected override void SetLeft(TLink node, TLink left) =>
        ↳ LinksIndexParts[node].LeftAsTarget = left;
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     protected override void SetRight(TLink node, TLink right) =>
        ↳ LinksIndexParts[node].RightAsTarget = right;
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected override void SetSize(TLink node, TLink size) =>
        ↳ LinksIndexParts[node].SizeAsTarget = size;
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override TLink GetTreeRoot() => Header->RootAsTarget;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↳ TLink secondSource, TLink secondTarget)
45         => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
        ↳ secondSource;
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↳ TLink secondSource, TLink secondTarget)
49         => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
        ↳ secondSource;
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override void ClearNode(TLink node)
53     {
54         ref var link = ref LinksIndexParts[node];
55         link.LeftAsTarget = Zero;
56         link.RightAsTarget = Zero;
57         link.SizeAsTarget = Zero;
58     }
59 }
60 }

```

#### 1.45 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSizeBalancedTreeMethodsBase.

```

1 using System.Runtime.CompilerServices;
2 using Platform.Data.Doublets.Memory.Split.Generic;
3 using TLink = System.UInt32;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Memory.Split.Specific
8 {
9     public unsafe abstract class UInt32InternalLinksSizeBalancedTreeMethodsBase :
        ↳ InternalLinksSizeBalancedTreeMethodsBase<TLink>
10     {
11         protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
12         protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;

```

```

13     protected new readonly LinksHeader<TLink>* Header;
14
15     [MethodImpl(MethodImplOptions.AggressiveInlining)]
16     protected UInt32InternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink>
17     ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
18     ↪ linksIndexParts, LinksHeader<TLink>* header)
19     : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
20     {
21         LinksDataParts = linksDataParts;
22         LinksIndexParts = linksIndexParts;
23         Header = header;
24     }
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected override TLink GetZero() => OU;
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     protected override bool EqualToZero(TLink value) => value == OU;
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     protected override bool AreEqual(TLink first, TLink second) => first == second;
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override bool GreaterThanZero(TLink value) => value > OU;
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override bool GreaterThan(TLink first, TLink second) => first > second;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override bool GreaterOrEqualThan(TLink first, TLink second) => first >= second;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override bool GreaterOrEqualThanZero(TLink value) => true; // value >= 0 is
46     ↪ always true for ulong
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected override bool LessOrEqualThanZero(TLink value) => value == OUL; // value is
50     ↪ always >= 0 for ulong
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override bool LessThanZero(TLink value) => false; // value < 0 is always false
57     ↪ for ulong
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override bool LessThan(TLink first, TLink second) => first < second;
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected override TLink Increment(TLink value) => ++value;
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override TLink Decrement(TLink value) => --value;
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     protected override TLink Add(TLink first, TLink second) => first + second;
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     protected override TLink Subtract(TLink first, TLink second) => first - second;
73
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
76     ↪ ref LinksDataParts[link];
77
78     [MethodImpl(MethodImplOptions.AggressiveInlining)]
79     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
80     ↪ ref LinksIndexParts[link];
81
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     protected override bool FirstIsToLeftOfSecond(TLink first, TLink second) =>
84     ↪ GetKeyPartValue(first) < GetKeyPartValue(second);
85
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
88     ↪ GetKeyPartValue(first) > GetKeyPartValue(second);
89
90 }
91 }

```

## 1.46 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesSizeBalancedTreeMethod

```

1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt32;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      public unsafe class UInt32InternalLinksSourcesSizeBalancedTreeMethods :
9          ↳ UInt32InternalLinksSizeBalancedTreeMethodsBase
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public UInt32InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
13             ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
14             ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
15             ↳ linksIndexParts, header) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected override ref TLink GetLeftReference(TLink node) => ref
19             ↳ LinksIndexParts[node].LeftAsSource;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override ref TLink GetRightReference(TLink node) => ref
23             ↳ LinksIndexParts[node].RightAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetLeft(TLink node, TLink left) =>
33             ↳ LinksIndexParts[node].LeftAsSource = left;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override void SetRight(TLink node, TLink right) =>
37             ↳ LinksIndexParts[node].RightAsSource = right;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override void SetSize(TLink node, TLink size) =>
44             ↳ LinksIndexParts[node].SizeAsSource = size;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override TLink GetTreeRoot(TLink node) => LinksIndexParts[node].RootAsSource;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Target;
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override void ClearNode(TLink node)
57         {
58             ref var link = ref LinksIndexParts[node];
59             link.LeftAsSource = Zero;
60             link.RightAsSource = Zero;
61             link.SizeAsSource = Zero;
62         }
63
64         public override TLink Search(TLink source, TLink target) =>
65             ↳ SearchCore(GetTreeRoot(source), target);
66     }
67 }

```

## 1.47 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksTargetsSizeBalancedTreeMethod

```

1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt32;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      public unsafe class UInt32InternalLinksTargetsSizeBalancedTreeMethods :
9          ↳ UInt32InternalLinksSizeBalancedTreeMethodsBase
10     {

```

```

10 [MethodImpl(MethodImplOptions.AggressiveInlining)]
11 public UInt32InternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink>
    ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
    ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
    ↳ linksIndexParts, header) { }

12
13 [MethodImpl(MethodImplOptions.AggressiveInlining)]
14 protected override ref TLink GetLeftReference(TLink node) => ref
    ↳ LinksIndexParts[node].LeftAsTarget;

15
16 [MethodImpl(MethodImplOptions.AggressiveInlining)]
17 protected override ref TLink GetRightReference(TLink node) => ref
    ↳ LinksIndexParts[node].RightAsTarget;

18
19 [MethodImpl(MethodImplOptions.AggressiveInlining)]
20 protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsTarget;

21
22 [MethodImpl(MethodImplOptions.AggressiveInlining)]
23 protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsTarget;

24
25 [MethodImpl(MethodImplOptions.AggressiveInlining)]
26 protected override void SetLeft(TLink node, TLink left) =>
    ↳ LinksIndexParts[node].LeftAsTarget = left;

27
28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 protected override void SetRight(TLink node, TLink right) =>
    ↳ LinksIndexParts[node].RightAsTarget = right;

30
31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;

33
34 [MethodImpl(MethodImplOptions.AggressiveInlining)]
35 protected override void SetSize(TLink node, TLink size) =>
    ↳ LinksIndexParts[node].SizeAsTarget = size;

36
37 [MethodImpl(MethodImplOptions.AggressiveInlining)]
38 protected override TLink GetTreeRoot(TLink node) => LinksIndexParts[node].RootAsTarget;

39
40 [MethodImpl(MethodImplOptions.AggressiveInlining)]
41 protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;

42
43 [MethodImpl(MethodImplOptions.AggressiveInlining)]
44 protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Source;

45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 protected override void ClearNode(TLink node)
48 {
49     ref var link = ref LinksIndexParts[node];
50     link.LeftAsTarget = Zero;
51     link.RightAsTarget = Zero;
52     link.SizeAsTarget = Zero;
53 }

54
55 public override TLink Search(TLink source, TLink target) =>
    ↳ SearchCore(GetTreeRoot(target), source);

56 }
57 }

```

#### 1.48 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32SplitMemoryLinks.cs

```

1 using System;
2 using System.Runtime.CompilerServices;
3 using Platform.Singletons;
4 using Platform.Memory;
5 using Platform.Data.Doublets.Memory.Split.Generic;
6 using TLink = System.UInt32;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Memory.Split.Specific
11 {
12     public unsafe class UInt32SplitMemoryLinks : SplitMemoryLinksBase<TLink>
13     {
14         private readonly Func<ILinksTreeMethods<TLink>> _createInternalSourceTreeMethods;
15         private readonly Func<ILinksTreeMethods<TLink>> _createExternalSourceTreeMethods;
16         private readonly Func<ILinksTreeMethods<TLink>> _createInternalTargetTreeMethods;
17         private readonly Func<ILinksTreeMethods<TLink>> _createExternalTargetTreeMethods;
18         private LinksHeader<TLink>* _header;
19         private RawLinkDataPart<TLink>* _linksDataParts;
20         private RawLinkIndexPart<TLink>* _linksIndexParts;
21

```

```

22 [MethodImpl(MethodImplOptions.AggressiveInlining)]
23 public UInt32SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }
24
25 [MethodImpl(MethodImplOptions.AggressiveInlining)]
26 public UInt32SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
    ↳ memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }
27
28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 public UInt32SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
    ↳ base(dataMemory, indexMemory, memoryReservationStep, constants)
30 {
31     _createInternalSourceTreeMethods = () => new
        ↳ UInt32InternalLinksSourcesSizeBalancedTreeMethods(Constants, _linksDataParts,
        ↳ _linksIndexParts, _header);
32     _createExternalSourceTreeMethods = () => new
        ↳ UInt32ExternalLinksSourcesSizeBalancedTreeMethods(Constants, _linksDataParts,
        ↳ _linksIndexParts, _header);
33     _createInternalTargetTreeMethods = () => new
        ↳ UInt32InternalLinksTargetsSizeBalancedTreeMethods(Constants, _linksDataParts,
        ↳ _linksIndexParts, _header);
34     _createExternalTargetTreeMethods = () => new
        ↳ UInt32ExternalLinksTargetsSizeBalancedTreeMethods(Constants, _linksDataParts,
        ↳ _linksIndexParts, _header);
35     Init(dataMemory, indexMemory);
36 }
37
38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 protected override void SetPointers(IResizableDirectMemory dataMemory,
    ↳ IResizableDirectMemory indexMemory)
40 {
41     _linksDataParts = (RawLinkDataPart<TLink>*)dataMemory.Pointer;
42     _linksIndexParts = (RawLinkIndexPart<TLink>*)indexMemory.Pointer;
43     _header = (LinksHeader<TLink>*)indexMemory.Pointer;
44     InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
45     ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
46     InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
47     ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
48     UnusedLinksListMethods = new UInt32UnusedLinksListMethods(_linksDataParts, _header);
49 }
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
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
    ↳ => ref _linksDataParts[linkIndex];
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
    ↳ linkIndex) => ref _linksIndexParts[linkIndex];
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected override bool AreEqual(TLink first, TLink second) => first == second;
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 protected override bool LessThan(TLink first, TLink second) => first < second;
74
75 [MethodImpl(MethodImplOptions.AggressiveInlining)]
76 protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
77
78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 protected override bool GreaterThan(TLink first, TLink second) => first > second;
80
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 protected override bool GreaterOrEqualThan(TLink first, TLink second) => first >= second;
83

```

```

84     [MethodImpl(MethodImplOptions.AggressiveInlining)]
85     protected override TLink GetZero() => 0U;
86
87     [MethodImpl(MethodImplOptions.AggressiveInlining)]
88     protected override TLink GetOne() => 1U;
89
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     protected override long ConvertToInt64(TLink value) => value;
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     protected override TLink ConvertToAddress(long value) => (TLink)value;
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     protected override TLink Add(TLink first, TLink second) => first + second;
98
99     [MethodImpl(MethodImplOptions.AggressiveInlining)]
100    protected override TLink Subtract(TLink first, TLink second) => first - second;
101
102    [MethodImpl(MethodImplOptions.AggressiveInlining)]
103    protected override TLink Increment(TLink link) => ++link;
104
105    [MethodImpl(MethodImplOptions.AggressiveInlining)]
106    protected override TLink Decrement(TLink link) => --link;
107 }
108 }

```

#### 1.49 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32UnusedLinksListMethods.cs

```

1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.Split.Generic;
3  using TLink = System.UInt32;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Memory.Split.Specific
8  {
9      public unsafe class UInt32UnusedLinksListMethods : UnusedLinksListMethods<TLink>
10     {
11         private readonly RawLinkDataPart<TLink>* _links;
12         private readonly LinksHeader<TLink>* _header;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public UInt32UnusedLinksListMethods(RawLinkDataPart<TLink>* links, LinksHeader<TLink>*
16             ↪ header)
17             : base((byte*)links, (byte*)header)
18         {
19             _links = links;
20             _header = header;
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
25             ↪ ref _links[link];
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
29     }
30 }

```

#### 1.50 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSizeBalancedTreeMethodsBase

```

1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.Split.Generic;
3  using TLink = System.UInt64;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Memory.Split.Specific
8  {
9      public unsafe abstract class UInt64ExternalLinksSizeBalancedTreeMethodsBase :
10         ↪ ExternalLinksSizeBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
11     {
12         protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
13         protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
14         protected new readonly LinksHeader<TLink>* Header;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected UInt64ExternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink>
18             ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
19             ↪ linksIndexParts, LinksHeader<TLink>* header)
20             : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
21         {
22
23         }
24     }
25 }

```

```

19     LinksDataParts = linksDataParts;
20     LinksIndexParts = linksIndexParts;
21     Header = header;
22 }
23
24 [MethodImpl(MethodImplOptions.AggressiveInlining)]
25 protected override ulong GetZero() => 0UL;
26
27 [MethodImpl(MethodImplOptions.AggressiveInlining)]
28 protected override bool EqualToZero(ulong value) => value == 0UL;
29
30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 protected override bool AreEqual(ulong first, ulong second) => first == second;
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 protected override bool GreaterThanZero(ulong value) => value > 0UL;
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 protected override bool GreaterThan(ulong first, ulong second) => first > second;
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
    ↳ always true for ulong
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
    ↳ always >= 0 for ulong
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
    ↳ for ulong
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected override bool LessThan(ulong first, ulong second) => first < second;
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 protected override ulong Increment(ulong value) => ++value;
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected override ulong Decrement(ulong value) => --value;
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected override ulong Add(ulong first, ulong second) => first + second;
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 protected override ulong Subtract(ulong first, ulong second) => first - second;
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected override ref LinkHeader<TLink> GetHeaderReference() => ref *Header;
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
    ↳ ref LinksDataParts[link];
74
75 [MethodImpl(MethodImplOptions.AggressiveInlining)]
76 protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
    ↳ ref LinksIndexParts[link];
77
78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 protected override bool FirstIsToLeftOfSecond(TLink first, TLink second)
80 {
81     ref var firstLink = ref LinksDataParts[first];
82     ref var secondLink = ref LinksDataParts[second];
83     return FirstIsToLeftOfSecond(firstLink.Source, firstLink.Target,
    ↳ secondLink.Source, secondLink.Target);
84 }
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
88 {
89     ref var firstLink = ref LinksDataParts[first];
90     ref var secondLink = ref LinksDataParts[second];

```



```

91         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
92             ↪ secondLink.Source, secondLink.Target);
93     }
94 }

```

## 1.51 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSourcesSizeBalancedTreeMethods

```

1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt64;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      public unsafe class UInt64ExternalLinksSourcesSizeBalancedTreeMethods :
9          ↪ UInt64ExternalLinksSizeBalancedTreeMethodsBase
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public UInt64ExternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
13             ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
14             ↪ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
15             ↪ linksIndexParts, header) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected override ref TLink GetLeftReference(TLink node) => ref
19             ↪ LinksIndexParts[node].LeftAsSource;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override ref TLink GetRightReference(TLink node) => ref
23             ↪ LinksIndexParts[node].RightAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetLeft(TLink node, TLink left) =>
33             ↪ LinksIndexParts[node].LeftAsSource = left;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override void SetRight(TLink node, TLink right) =>
37             ↪ LinksIndexParts[node].RightAsSource = right;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override void SetSize(TLink node, TLink size) =>
44             ↪ LinksIndexParts[node].SizeAsSource = size;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override TLink GetTreeRoot() => Header->RootAsSource;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
54             ↪ TLink secondSource, TLink secondTarget)
55             => firstSource < secondSource || firstSource == secondSource && firstTarget <
56             ↪ secondTarget;
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
60             ↪ TLink secondSource, TLink secondTarget)
61             => firstSource > secondSource || firstSource == secondSource && firstTarget >
62             ↪ secondTarget;
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected override void ClearNode(TLink node)
66         {
67             ref var link = ref LinksIndexParts[node];
68             link.LeftAsSource = Zero;
69             link.RightAsSource = Zero;
70             link.SizeAsSource = Zero;
71         }
72     }
73 }

```

```
60 }
```

## 1.52 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksTargetsSizeBalancedTreeMethods

```
1 using System.Runtime.CompilerServices;
2 using TLink = System.UInt64;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Memory.Split.Specific
7 {
8     public unsafe class UInt64ExternalLinksTargetsSizeBalancedTreeMethods :
9         ↳ UInt64ExternalLinksSizeBalancedTreeMethodsBase
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public UInt64ExternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink>
13             ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
14             ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
15             ↳ linksIndexParts, header) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected override ref TLink GetLeftReference(TLink node) => ref
19             ↳ LinksIndexParts[node].LeftAsTarget;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override ref TLink GetRightReference(TLink node) => ref
23             ↳ LinksIndexParts[node].RightAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsTarget;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsTarget;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetLeft(TLink node, TLink left) =>
33             ↳ LinksIndexParts[node].LeftAsTarget = left;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override void SetRight(TLink node, TLink right) =>
37             ↳ LinksIndexParts[node].RightAsTarget = right;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override void SetSize(TLink node, TLink size) =>
44             ↳ LinksIndexParts[node].SizeAsTarget = size;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override TLink GetTreeRoot() => Header->RootAsTarget;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
54             ↳ TLink secondSource, TLink secondTarget)
55             => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
56             ↳ secondSource;
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
60             ↳ TLink secondSource, TLink secondTarget)
61             => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
62             ↳ secondSource;
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected override void ClearNode(TLink node)
66         {
67             ref var link = ref LinksIndexParts[node];
68             link.LeftAsTarget = Zero;
69             link.RightAsTarget = Zero;
70             link.SizeAsTarget = Zero;
71         }
72     }
73 }
```

1.53 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSizeBalancedTreeMethodsBase.

```
1 using System.Runtime.CompilerServices;
2 using Platform.Data.Doublets.Memory.Split.Generic;
3 using TLink = System.UInt64;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Memory.Split.Specific
8 {
9     public unsafe abstract class UInt64InternalLinksSizeBalancedTreeMethodsBase :
10     ↪ InternalLinksSizeBalancedTreeMethodsBase<TLink>
11     {
12         protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
13         protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
14         protected new readonly LinksHeader<TLink>* Header;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected UInt64InternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink>
18         ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
19         ↪ linksIndexParts, LinksHeader<TLink>* header)
20             : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
21         {
22             LinksDataParts = linksDataParts;
23             LinksIndexParts = linksIndexParts;
24             Header = header;
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override ulong GetZero() => OUL;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override bool EqualToZero(ulong value) => value == OUL;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override bool AreEqual(ulong first, ulong second) => first == second;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override bool GreaterThanZero(ulong value) => value > OUL;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override bool GreaterThan(ulong first, ulong second) => first > second;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
47         ↪ always true for ulong
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool LessOrEqualThanZero(ulong value) => value == OUL; // value is
51         ↪ always >= 0 for ulong
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
58         ↪ for ulong
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         protected override bool LessThan(ulong first, ulong second) => first < second;
62
63         [MethodImpl(MethodImplOptions.AggressiveInlining)]
64         protected override ulong Increment(ulong value) => ++value;
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         protected override ulong Decrement(ulong value) => --value;
68
69         [MethodImpl(MethodImplOptions.AggressiveInlining)]
70         protected override ulong Add(ulong first, ulong second) => first + second;
71
72         [MethodImpl(MethodImplOptions.AggressiveInlining)]
73         protected override ulong Subtract(ulong first, ulong second) => first - second;
74
75         [MethodImpl(MethodImplOptions.AggressiveInlining)]
76         protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
77         ↪ ref LinksDataParts[link];
78
79         [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     }
```

```

73     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
74         ↪ ref LinksIndexParts[link];
75
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
78         ↪ GetKeyPartValue(first) < GetKeyPartValue(second);
79
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
82         ↪ GetKeyPartValue(first) > GetKeyPartValue(second);
83 }
84 }

```

#### 1.54 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesSizeBalancedTreeMethod

```

1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt64;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      public unsafe class UInt64InternalLinksSourcesSizeBalancedTreeMethods :
9          ↪ UInt64InternalLinksSizeBalancedTreeMethodsBase
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public UInt64InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
13             ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
14             ↪ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
15             ↪ linksIndexParts, header) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected override ref TLink GetLeftReference(TLink node) => ref
19             ↪ LinksIndexParts[node].LeftAsSource;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override ref TLink GetRightReference(TLink node) => ref
23             ↪ LinksIndexParts[node].RightAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetLeft(TLink node, TLink left) =>
33             ↪ LinksIndexParts[node].LeftAsSource = left;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override void SetRight(TLink node, TLink right) =>
37             ↪ LinksIndexParts[node].RightAsSource = right;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override void SetSize(TLink node, TLink size) =>
44             ↪ LinksIndexParts[node].SizeAsSource = size;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override TLink GetTreeRoot(TLink node) => LinksIndexParts[node].RootAsSource;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Target;
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override void ClearNode(TLink node)
57         {
58             ref var link = ref LinksIndexParts[node];
59             link.LeftAsSource = Zero;
60             link.RightAsSource = Zero;
61             link.SizeAsSource = Zero;
62         }
63
64         public override TLink Search(TLink source, TLink target) =>
65             ↪ SearchCore(GetTreeRoot(source), target);
66     }
67 }

```

```

56     }
57 }

```

## 1.55 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksTargetsSizeBalancedTreeMethods

```

1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt64;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      public unsafe class UInt64InternalLinksTargetsSizeBalancedTreeMethods :
9          ↳ UInt64InternalLinksSizeBalancedTreeMethodsBase
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public UInt64InternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink>
13             ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
14             ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
15             ↳ linksIndexParts, header) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected override ref ulong GetLeftReference(ulong node) => ref
19             ↳ LinksIndexParts[node].LeftAsTarget;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override ref ulong GetRightReference(ulong node) => ref
23             ↳ LinksIndexParts[node].RightAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsTarget;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsTarget;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetLeft(TLink node, TLink left) =>
33             ↳ LinksIndexParts[node].LeftAsTarget = left;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override void SetRight(TLink node, TLink right) =>
37             ↳ LinksIndexParts[node].RightAsTarget = right;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override void SetSize(TLink node, TLink size) =>
44             ↳ LinksIndexParts[node].SizeAsTarget = size;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override TLink GetTreeRoot(TLink node) => LinksIndexParts[node].RootAsTarget;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Source;
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override void ClearNode(TLink node)
57         {
58             ref var link = ref LinksIndexParts[node];
59             link.LeftAsTarget = Zero;
60             link.RightAsTarget = Zero;
61             link.SizeAsTarget = Zero;
62         }
63
64         public override TLink Search(TLink source, TLink target) =>
65             ↳ SearchCore(GetTreeRoot(target), source);
66     }
67 }

```

## 1.56 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64SplitMemoryLinks.cs

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using Platform.Data.Doublets.Memory.Split.Generic;
6  using TLink = System.UInt64;

```

```

7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Memory.Split.Specific
11 {
12     public unsafe class UInt64SplitMemoryLinks : SplitMemoryLinksBase<TLink>
13     {
14         private readonly Func<ILinksTreeMethods<TLink>> _createInternalSourceTreeMethods;
15         private readonly Func<ILinksTreeMethods<TLink>> _createExternalSourceTreeMethods;
16         private readonly Func<ILinksTreeMethods<TLink>> _createInternalTargetTreeMethods;
17         private readonly Func<ILinksTreeMethods<TLink>> _createExternalTargetTreeMethods;
18         private LinksHeader<ulong>* _header;
19         private RawLinkDataPart<ulong>* _linksDataParts;
20         private RawLinkIndexPart<ulong>* _linksIndexParts;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
24             ↪ indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
28             ↪ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
29             ↪ memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
33             ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
34             ↪ base(dataMemory, indexMemory, memoryReservationStep, constants)
35         {
36             _createInternalSourceTreeMethods = () => new
37                 ↪ UInt64InternalLinksSourcesSizeBalancedTreeMethods(Constants, _linksDataParts,
38                 ↪ _linksIndexParts, _header);
39             _createExternalSourceTreeMethods = () => new
40                 ↪ UInt64ExternalLinksSourcesSizeBalancedTreeMethods(Constants, _linksDataParts,
41                 ↪ _linksIndexParts, _header);
42             _createInternalTargetTreeMethods = () => new
43                 ↪ UInt64InternalLinksTargetsSizeBalancedTreeMethods(Constants, _linksDataParts,
44                 ↪ _linksIndexParts, _header);
45             _createExternalTargetTreeMethods = () => new
46                 ↪ UInt64ExternalLinksTargetsSizeBalancedTreeMethods(Constants, _linksDataParts,
47                 ↪ _linksIndexParts, _header);
48             Init(dataMemory, indexMemory);
49         }
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override void SetPointers(IResizableDirectMemory dataMemory,
53             ↪ IResizableDirectMemory indexMemory)
54         {
55             _linksDataParts = (RawLinkDataPart<TLink>*)dataMemory.Pointer;
56             _linksIndexParts = (RawLinkIndexPart<TLink>*)indexMemory.Pointer;
57             _header = (LinksHeader<TLink>*)indexMemory.Pointer;
58             InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
59             ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
60             InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
61             ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
62             UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_linksDataParts, _header);
63         }
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         protected override void ResetPointers()
67         {
68             base.ResetPointers();
69             _linksDataParts = null;
70             _linksIndexParts = null;
71             _header = null;
72         }
73
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]
75         protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
76
77         [MethodImpl(MethodImplOptions.AggressiveInlining)]
78         protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
79             ↪ => ref _linksDataParts[linkIndex];
80
81         [MethodImpl(MethodImplOptions.AggressiveInlining)]
82         protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
83             ↪ linkIndex) => ref _linksIndexParts[linkIndex];
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

```

69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override bool AreEqual(ulong first, ulong second) => first == second;
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     protected override bool LessThan(ulong first, ulong second) => first < second;
74
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
77
78     [MethodImpl(MethodImplOptions.AggressiveInlining)]
79     protected override bool GreaterThan(ulong first, ulong second) => first > second;
80
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
83
84     [MethodImpl(MethodImplOptions.AggressiveInlining)]
85     protected override ulong GetZero() => 0UL;
86
87     [MethodImpl(MethodImplOptions.AggressiveInlining)]
88     protected override ulong GetOne() => 1UL;
89
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     protected override long ConvertToInt64(ulong value) => (long)value;
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     protected override ulong ConvertToAddress(long value) => (ulong)value;
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     protected override ulong Add(ulong first, ulong second) => first + second;
98
99     [MethodImpl(MethodImplOptions.AggressiveInlining)]
100    protected override ulong Subtract(ulong first, ulong second) => first - second;
101
102    [MethodImpl(MethodImplOptions.AggressiveInlining)]
103    protected override ulong Increment(ulong link) => ++link;
104
105    [MethodImpl(MethodImplOptions.AggressiveInlining)]
106    protected override ulong Decrement(ulong link) => --link;
107 }
108 }

```

### 1.57 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64UnusedLinksListMethods.cs

```

1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.Split.Generic;
3  using TLink = System.UInt64;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Memory.Split.Specific
8  {
9      public unsafe class UInt64UnusedLinksListMethods : UnusedLinksListMethods<TLink>
10     {
11         private readonly RawLinkDataPart<ulong>* _links;
12         private readonly LinksHeader<ulong>* _header;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public UInt64UnusedLinksListMethods(RawLinkDataPart<ulong>* links, LinksHeader<ulong>*
16             ↪ header)
17             : base((byte*)links, (byte*)header)
18         {
19             _links = links;
20             _header = header;
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
25             ↪ ref _links[link];
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
29     }
30 }

```

### 1.58 ./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> :
15         ↳ SizedAndThreadedAVLBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
16     {
17         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
18             ↳ UncheckedConverter<TLink, long>.Default;
19         private static readonly UncheckedConverter<TLink, int> _addressToInt32Converter =
20             ↳ UncheckedConverter<TLink, int>.Default;
21         private static readonly UncheckedConverter<bool, TLink> _boolToAddressConverter =
22             ↳ UncheckedConverter<bool, TLink>.Default;
23         private static readonly UncheckedConverter<TLink, bool> _addressToBoolConverter =
24             ↳ UncheckedConverter<TLink, bool>.Default;
25         private static readonly UncheckedConverter<int, TLink> _int32ToAddressConverter =
26             ↳ UncheckedConverter<int, TLink>.Default;
27
28         protected readonly TLink Break;
29         protected readonly TLink Continue;
30         protected readonly byte* Links;
31         protected readonly byte* Header;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
35             ↳ byte* header)
36         {
37             Links = links;
38             Header = header;
39             Break = constants.Break;
40             Continue = constants.Continue;
41         }
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected abstract TLink GetTreeRoot();
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected abstract TLink GetBasePartValue(TLink link);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
51             ↳ rootSource, TLink rootTarget);
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
55             ↳ rootSource, TLink rootTarget);
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
59             ↳ AsRef<LinksHeader<TLink>>(Header);
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
63             ↳ AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
64             ↳ _addressToInt64Converter.Convert(link)));
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
68         {
69             ref var link = ref GetLinkReference(linkIndex);
70             return new Link<TLink>(linkIndex, link.Source, link.Target);
71         }
72
73         [MethodImpl(MethodImplOptions.AggressiveInlining)]
74         protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
75         {
76             ref var firstLink = ref GetLinkReference(first);
77             ref var secondLink = ref GetLinkReference(second);
78             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
79                 ↳ secondLink.Source, secondLink.Target);
80         }
81
82         [MethodImpl(MethodImplOptions.AggressiveInlining)]
83         protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
84         {
85             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,
    ↪ _boolToAddressConverter.Convert(value), 3, 1);
121         storedValue = modified;
122     }
123 }
124
125 [MethodImpl(MethodImplOptions.AggressiveInlining)]
126 protected bool IsChild(TLink parent, TLink possibleChild)
127 {
128     var parentSize = GetSize(parent);
129     var childSize = GetSizeOrZero(possibleChild);
130     return GreaterThanZero(childSize) && LessOrEqualThan(childSize, parentSize);
131 }
132
133 [MethodImpl(MethodImplOptions.AggressiveInlining)]
134 protected virtual sbyte GetBalanceValue(TLink storedValue)
135 {
136     unchecked
137     {
138         var value = _addressToInt32Converter.Convert(Bit<TLink>.PartialRead(storedValue,
    ↪ 0, 3));
139         value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
    ↪ end of sbyte
140         return (sbyte)value;
141     }
142 }
143
144 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

145 protected virtual void SetBalanceValue(ref TLink storedValue, sbyte value)
146 {
147     unchecked
148     {
149         var packagedValue = _int32ToAddressConverter.Convert((byte)value >> 5 & 4 |
150             ↪ value & 3);
151         var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
152         storedValue = modified;
153     }
154 }
155 public TLink this[TLink index]
156 {
157     [MethodImpl(MethodImplOptions.AggressiveInlining)]
158     get
159     {
160         var root = GetTreeRoot();
161         if (GreaterOrEqualThan(index, GetSize(root)))
162         {
163             return Zero;
164         }
165         while (!EqualToZero(root))
166         {
167             var left = GetLeftOrDefault(root);
168             var leftSize = GetSizeOrZero(left);
169             if (LessThan(index, leftSize))
170             {
171                 root = left;
172                 continue;
173             }
174             if (AreEqual(index, leftSize))
175             {
176                 return root;
177             }
178             root = GetRightOrDefault(root);
179             index = Subtract(index, Increment(leftSize));
180         }
181         return Zero; // TODO: Impossible situation exception (only if tree structure
182             ↪ broken)
183     }
184 }
185 /// <summary>
186 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
187   ↪ (концом).
188 /// </summary>
189 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
190 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
191 /// <returns>Индекс искомой связи.</returns>
192 [MethodImpl(MethodImplOptions.AggressiveInlining)]
193 public TLink Search(TLink source, TLink target)
194 {
195     var root = GetTreeRoot();
196     while (!EqualToZero(root))
197     {
198         ref var rootLink = ref GetLinkReference(root);
199         var rootSource = rootLink.Source;
200         var rootTarget = rootLink.Target;
201         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
202             ↪ node.Key < root.Key
203         {
204             root = GetLeftOrDefault(root);
205         }
206         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
207             ↪ node.Key > root.Key
208         {
209             root = GetRightOrDefault(root);
210         }
211         else // node.Key == root.Key
212         {
213             return root;
214         }
215     }
216     return Zero;
217 }
218 // TODO: Return indices range instead of references count
219 [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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
255 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
256 {
257     var root = GetTreeRoot();
258     if (EqualToZero(root))
259     {
260         return Continue;
261     }
262     TLink first = Zero, current = root;
263     while (!EqualToZero(current))
264     {
265         var @base = GetBasePartValue(current);
266         if (GreaterOrEqualThan(@base, link))
267         {
268             if (AreEqual(@base, link))
269             {
270                 first = current;
271             }
272             current = GetLeftOrDefault(current);
273         }
274         else
275         {
276             current = GetRightOrDefault(current);
277         }
278     }
279     if (!EqualToZero(first))
280     {
281         current = first;
282         while (true)
283         {
284             if (AreEqual(handler(GetLinkValues(current)), Break))
285             {
286                 return Break;
287             }
288             current = GetNext(current);
289             if (EqualToZero(current) || !AreEqual(GetBasePartValue(current), link))
290             {
291                 break;
292             }
293         }
294     }
295     return Continue;
296 }

```

```

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 }

```

## 1.59 ./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;
21         protected readonly byte* Links;
22         protected readonly byte* Header;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
26             ↳ byte* header)
27         {
28             Links = links;
29             Header = header;
30             Break = constants.Break;
31             Continue = constants.Continue;
32         }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected abstract TLink GetTreeRoot();
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected abstract TLink GetBasePartValue(TLink link);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
42             ↳ rootSource, TLink rootTarget);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
46             ↳ rootSource, TLink rootTarget);
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
50             ↳ AsRef<LinksHeader<TLink>>(Header);
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
54             ↳ AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
55                 ↳ _addressToInt64Converter.Convert(link)));
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
59         {
60             ref var link = ref GetLinkReference(linkIndex);
61             return new Link<TLink>(linkIndex, link.Source, link.Target);
62         }
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

57     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
58     {
59         ref var firstLink = ref GetLinkReference(first);
60         ref var secondLink = ref GetLinkReference(second);
61         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
        ↪ secondLink.Source, secondLink.Target);
62     }
63
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
66     {
67         ref var firstLink = ref GetLinkReference(first);
68         ref var secondLink = ref GetLinkReference(second);
69         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
        ↪ secondLink.Source, secondLink.Target);
70     }
71
72     public TLink this[TLink index]
73     {
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]
75         get
76         {
77             var root = GetTreeRoot();
78             if (GreaterOrEqualThan(index, GetSize(root)))
79             {
80                 return Zero;
81             }
82             while (!EqualToZero(root))
83             {
84                 var left = GetLeftOrDefault(root);
85                 var leftSize = GetSizeOrZero(left);
86                 if (LessThan(index, leftSize))
87                 {
88                     root = left;
89                     continue;
90                 }
91                 if (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         }

```

```

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))

```



```

44 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 protected override void SetRightIsChild(TLink node, bool value) =>
46     ↳ SetRightIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 protected override sbyte GetBalance(TLink node) =>
50     ↳ GetBalanceValue(GetLinkReference(node).SizeAsSource);
51
52 [MethodImpl(MethodImplOptions.AggressiveInlining)]
53 protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
54     ↳ GetLinkReference(node).SizeAsSource, value);
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
64     ↳ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
65     ↳ (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
69     ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
70     ↳ (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 protected override void ClearNode(TLink node)
74 {
75     ref var link = ref GetLinkReference(node);
76     link.LeftAsSource = Zero;
77     link.RightAsSource = Zero;
78     link.SizeAsSource = Zero;
79 }
80 }
81 }

```

## 1.61 ./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 }

```



```

33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override void SetSize(TLink node, TLink size) =>
        ↳ GetLinkReference(node).SizeAsSource = size;
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↳ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
        ↳ (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
        ↳ (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected override void ClearNode(TLink node)
50     {
51         ref var link = ref GetLinkReference(node);
52         link.LeftAsSource = Zero;
53         link.RightAsSource = Zero;
54         link.SizeAsSource = Zero;
55     }
56 }
57 }

```

## 1.62 ./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> :
        ↳ LinksAvlBalancedTreeMethodsBase<TLink>
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public LinksTargetsAvlBalancedTreeMethods(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) =>
            ↳ GetSizeValue(GetLinkReference(node).SizeAsTarget);
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
            ↳ GetLinkReference(node).SizeAsTarget, size);
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override bool GetLeftIsChild(TLink node) =>
            ↳ GetLeftIsChildValue(GetLinkReference(node).SizeAsTarget);
38

```

```

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

```

### 1.63 ./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> :
8          ↪ LinksSizeBalancedTreeMethodsBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksTargetsSizeBalancedTreeMethods(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 }

```

```

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 }

```

#### 1.64 ./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
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
    ↪ FileMappedResizableDirectMemory(address, memoryReservationStep),
    ↪ memoryReservationStep) { }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
    ↪ DefaultLinksSizeStep) { }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep) :
    ↪ this(memory, memoryReservationStep, Default<LinksConstants<TLink>>.Instance,
    ↪ IndexTreeType.Default) { }
37     }
38 }

```

```

34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep,
    ↳ LinksConstants<TLink> constants, IndexTreeType indexTreeType) : base(memory,
    ↳ memoryReservationStep, constants)
37 {
38     if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
39     {
40         _createSourceTreeMethods = () => new
            ↳ LinksSourcesAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
41         _createTargetTreeMethods = () => new
            ↳ LinksTargetsAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
42     }
43     else
44     {
45         _createSourceTreeMethods = () => new
            ↳ LinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
46         _createTargetTreeMethods = () => new
            ↳ LinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
47     }
48     Init(memory, memoryReservationStep);
49 }
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override void SetPointers(IResizableDirectMemory memory)
53 {
54     _links = (byte*)memory.Pointer;
55     _header = _links;
56     SourcesTreeMethods = _createSourceTreeMethods();
57     TargetsTreeMethods = _createTargetTreeMethods();
58     UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_links, _header);
59 }
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected override void ResetPointers()
63 {
64     base.ResetPointers();
65     _links = null;
66     _header = null;
67 }
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected override ref LinksHeader<TLink> GetHeaderReference() => ref
    ↳ AsRef<LinksHeader<TLink>>(_header);
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 protected override ref RawLink<TLink> GetLinkReference(TLink linkIndex) => ref
    ↳ AsRef<RawLink<TLink>>(_links + (LinkSizeInBytes * ConvertToInt64(linkIndex)));
74 }
75 }

```

## 1.65 ./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 =
            ↳ EqualityComparer<TLink>.Default;
18         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
19         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            ↳ UncheckedConverter<TLink, long>.Default;
20         private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
            ↳ UncheckedConverter<long, TLink>.Default;
21
22         private static readonly TLink _zero = default;
23         private static readonly TLink _one = Arithmetic.Increment(_zero);
24

```

```

25     /// <summary>Возвращает размер одной связи в байтах.</summary>
26     /// <remarks>
27     /// Используется только во вне класса, не рекомендуется использовать внутри.
28     /// Так как во вне не обязательно будет доступен unsafe C#.
29     /// </remarks>
30     public static readonly long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;
31
32     public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
33
34     public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
35
36     protected readonly IResizableDirectMemory _memory;
37     protected readonly long _memoryReservationStep;
38
39     protected ILinksTreeMethods<TLink> TargetsTreeMethods;
40     protected ILinksTreeMethods<TLink> SourcesTreeMethods;
41     // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
42     // → нужно использовать не список а дерево, так как так можно быстрее проверить на
43     // → наличие связи внутри
44     protected ILinksListMethods<TLink> UnusedLinksListMethods;
45
46     /// <summary>
47     /// Возвращает общее число связей находящихся в хранилище.
48     /// </summary>
49     protected virtual TLink Total
50     {
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         get
53         {
54             ref var header = ref GetHeaderReference();
55             return Subtract(header.AllocatedLinks, header.FreeLinks);
56         }
57     }
58
59     public virtual LinksConstants<TLink> Constants
60     {
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         get;
63     }
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
67     → memoryReservationStep, LinksConstants<TLink> constants)
68     {
69         _memory = memory;
70         _memoryReservationStep = memoryReservationStep;
71         Constants = constants;
72     }
73
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
76     → memoryReservationStep) : this(memory, memoryReservationStep,
77     → Default<LinksConstants<TLink>>.Instance) { }
78
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
81     {
82         if (memory.ReservedCapacity < memoryReservationStep)
83         {
84             memory.ReservedCapacity = memoryReservationStep;
85         }
86         SetPointers(memory);
87         ref var header = ref GetHeaderReference();
88         // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
89         memory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) * LinkSizeInBytes) +
90         → LinkHeaderSizeInBytes;
91         // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
92         header.ReservedLinks = ConvertToAddress((memory.ReservedCapacity -
93         → LinkHeaderSizeInBytes) / LinkSizeInBytes);
94     }
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     public virtual TLink Count(IList<TLink> restrictions)
98     {
99         // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
100         if (restrictions.Count == 0)
101         {
102             return Total;
103         }
104         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),
118             ↪ TargetsTreeMethods.CountUsages(value));
119     }
120     else
121     {
122         if (!Exists(index))
123         {
124             return GetZero();
125         }
126         if (AreEqual(value, any))
127         {
128             return GetOne();
129         }
130         ref var storedLinkValue = ref GetLinkReference(index);
131         if (AreEqual(storedLinkValue.Source, value) ||
132             ↪ AreEqual(storedLinkValue.Target, value))
133         {
134             return GetOne();
135         }
136         return GetZero();
137     }
138 }
139 if (restrictions.Count == 3)
140 {
141     var source = restrictions[constants.SourcePart];
142     var target = restrictions[constants.TargetPart];
143     if (AreEqual(index, any))
144     {
145         if (AreEqual(source, any) && AreEqual(target, any))
146         {
147             return Total;
148         }
149         else if (AreEqual(source, any))
150         {
151             return TargetsTreeMethods.CountUsages(target);
152         }
153         else if (AreEqual(target, any))
154         {
155             return SourcesTreeMethods.CountUsages(source);
156         }
157         else //if(source != Any && target != Any)
158         {
159             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
160             var link = SourcesTreeMethods.Search(source, target);
161             return AreEqual(link, constants.Null) ? GetZero() : GetOne();
162         }
163     }
164     else
165     {
166         if (!Exists(index))
167         {
168             return GetZero();
169         }
170         if (AreEqual(source, any) && AreEqual(target, any))
171         {
172             return GetOne();
173         }
174         ref var storedLinkValue = ref GetLinkReference(index);
175         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,
                ↪ GetHeaderReference().AllocatedLinks); link = Increment(link))
208         {
209             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
210             {
211                 return @break;
212             }
213         }
214         return @break;
215     }
216     var @continue = constants.Continue;
217     var any = constants.Any;
218     var index = restrictions[constants.IndexPart];
219     if (restrictions.Count == 1)
220     {
221         if (AreEqual(index, any))
222         {
223             return Each(handler, Array.Empty<TLink>());
224         }
225         if (!Exists(index))
226         {
227             return @continue;
228         }
229         return handler(GetLinkStruct(index));
230     }
231     if (restrictions.Count == 2)
232     {
233         var value = restrictions[1];
234         if (AreEqual(index, any))
235         {
236             if (AreEqual(value, any))
237             {
238                 return Each(handler, Array.Empty<TLink>());
239             }
240             if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
241             {
242                 return @break;
243             }
244             return Each(handler, new Link<TLink>(index, any, value));
245         }
246         else
247         {

```

```

248         if (!Exists(index))
249         {
250             return @continue;
251         }
252         if (AreEqual(value, any))
253         {
254             return handler(GetLinkStruct(index));
255         }
256         ref var storedLinkValue = ref GetLinkReference(index);
257         if (AreEqual(storedLinkValue.Source, value) ||
258             AreEqual(storedLinkValue.Target, value))
259         {
260             return handler(GetLinkStruct(index));
261         }
262         return @continue;
263     }
264 }
265 if (restrictions.Count == 3)
266 {
267     var source = restrictions[constants.SourcePart];
268     var target = restrictions[constants.TargetPart];
269     if (AreEqual(index, any))
270     {
271         if (AreEqual(source, any) && AreEqual(target, any))
272         {
273             return Each(handler, Array.Empty<TLink>());
274         }
275         else if (AreEqual(source, any))
276         {
277             return TargetsTreeMethods.EachUsage(target, handler);
278         }
279         else if (AreEqual(target, any))
280         {
281             return SourcesTreeMethods.EachUsage(source, handler);
282         }
283         else //if(source != Any && target != Any)
284         {
285             var link = SourcesTreeMethods.Search(source, target);
286             return AreEqual(link, constants.Null) ? @continue :
287                 ↪ handler(GetLinkStruct(link));
288         }
289     }
290     else
291     {
292         if (!Exists(index))
293         {
294             return @continue;
295         }
296         if (AreEqual(source, any) && AreEqual(target, any))
297         {
298             return handler(GetLinkStruct(index));
299         }
300         ref var storedLinkValue = ref GetLinkReference(index);
301         if (!AreEqual(source, any) && !AreEqual(target, any))
302         {
303             if (AreEqual(storedLinkValue.Source, source) &&
304                 AreEqual(storedLinkValue.Target, target))
305             {
306                 return handler(GetLinkStruct(index));
307             }
308             return @continue;
309         }
310         var value = default(TLink);
311         if (AreEqual(source, any))
312         {
313             value = target;
314         }
315         if (AreEqual(target, any))
316         {
317             value = source;
318         }
319         if (AreEqual(storedLinkValue.Source, value) ||
320             AreEqual(storedLinkValue.Target, value))
321         {
322             return handler(GetLinkStruct(index));
323         }
324         return @continue;
325     }
326 }

```



```

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(ICollection<TLink> restrictions, ICollection<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
    ↳ пространство
366 /// </remarks>
367 [MethodImpl(MethodImplOptions.AggressiveInlining)]
368 public virtual TLink Create(ICollection<TLink> restrictions)
369 {
370     ref var header = ref GetHeaderReference();
371     var freeLink = header.FirstFreeLink;
372     if (!AreEqual(freeLink, Constants.Null))
373     {
374         UnusedLinksListMethods.Detach(freeLink);
375     }
376     else
377     {
378         var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
379         if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
380         {
381             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
382         }
383         if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
384         {
385             _memory.ReservedCapacity += _memory.ReservationStep;
386             SetPointers(_memory);
387             header = ref GetHeaderReference();
388             header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
    ↳ LinkSizeInBytes);
389         }
390         header.AllocatedLinks = Increment(header.AllocatedLinks);
391         _memory.UsedCapacity += LinkSizeInBytes;
392         freeLink = header.AllocatedLinks;
393     }
394     return freeLink;
395 }
396
397 [MethodImpl(MethodImplOptions.AggressiveInlining)]
398 public virtual void Delete(ICollection<TLink> restrictions)

```

```

399 {
400     ref var header = ref GetHeaderReference();
401     var link = restrictions[Constants.IndexPart];
402     if (LessThan(link, header.AllocatedLinks))
403     {
404         UnusedLinksListMethods.AttachAsFirst(link);
405     }
406     else if (AreEqual(link, header.AllocatedLinks))
407     {
408         header.AllocatedLinks = Decrement(header.AllocatedLinks);
409         _memory.UsedCapacity -= LinkSizeInBytes;
410         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
411         // ↳ пока не дойдём до первой существующей связи
412         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
413         while (GreaterThan(header.AllocatedLinks, GetZero()) &&
414             ↳ IsUnusedLink(header.AllocatedLinks))
415         {
416             UnusedLinksListMethods.Detach(header.AllocatedLinks);
417             header.AllocatedLinks = Decrement(header.AllocatedLinks);
418             _memory.UsedCapacity -= LinkSizeInBytes;
419         }
420     }
421 }
422 [MethodImpl(MethodImplOptions.AggressiveInlining)]
423 public IList<TLink> GetLinkStruct(TLink linkIndex)
424 {
425     ref var link = ref GetLinkReference(linkIndex);
426     return new Link<TLink>(linkIndex, link.Source, link.Target);
427 }
428
429 /// <remarks>
430 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
431 /// ↳ адрес реально поменялся
432 ///
433 /// Указатель this.links может быть в том же месте,
434 /// так как 0-я связь не используется и имеет такой же размер как Header,
435 /// поэтому header размещается в том же месте, что и 0-я связь
436 /// </remarks>
437 [MethodImpl(MethodImplOptions.AggressiveInlining)]
438 protected abstract void SetPointers(IResizableDirectMemory memory);
439
440 [MethodImpl(MethodImplOptions.AggressiveInlining)]
441 protected virtual void ResetPointers()
442 {
443     SourcesTreeMethods = null;
444     TargetsTreeMethods = null;
445     UnusedLinksListMethods = null;
446 }
447
448 [MethodImpl(MethodImplOptions.AggressiveInlining)]
449 protected abstract ref LinksHeader<TLink> GetHeaderReference();
450
451 [MethodImpl(MethodImplOptions.AggressiveInlining)]
452 protected abstract ref RawLink<TLink> GetLinkReference(TLink linkIndex);
453
454 [MethodImpl(MethodImplOptions.AggressiveInlining)]
455 protected virtual bool Exists(TLink link)
456 => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
457 && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
458 && !IsUnusedLink(link);
459
460 [MethodImpl(MethodImplOptions.AggressiveInlining)]
461 protected virtual bool IsUnusedLink(TLink linkIndex)
462 {
463     if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
464     ↳ is not needed
465     {
466         ref var link = ref GetLinkReference(linkIndex);
467         return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
468     }
469     else
470     {
471         return true;
472     }
473 }
474
475 [MethodImpl(MethodImplOptions.AggressiveInlining)]
476 protected virtual TLink GetOne() => _one;

```

```

474 [MethodImpl(MethodImplOptions.AggressiveInlining)]
475 protected virtual TLink GetZero() => default;
476
477 [MethodImpl(MethodImplOptions.AggressiveInlining)]
478 protected virtual bool AreEqual(TLink first, TLink second) =>
479     ↪ _equalityComparer.Equals(first, second);
480
481 [MethodImpl(MethodImplOptions.AggressiveInlining)]
482 protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
483     ↪ second) < 0;
484
485 [MethodImpl(MethodImplOptions.AggressiveInlining)]
486 protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
487     ↪ _comparer.Compare(first, second) <= 0;
488
489 [MethodImpl(MethodImplOptions.AggressiveInlining)]
490 protected virtual bool GreaterThan(TLink first, TLink second) =>
491     ↪ _comparer.Compare(first, second) > 0;
492
493 [MethodImpl(MethodImplOptions.AggressiveInlining)]
494 protected virtual long ConvertToInt64(TLink value) =>
495     ↪ _addressToInt64Converter.Convert(value);
496
497 [MethodImpl(MethodImplOptions.AggressiveInlining)]
498 protected virtual TLink ConvertToAddress(long value) =>
499     ↪ _int64ToAddressConverter.Convert(value);
500
501 [MethodImpl(MethodImplOptions.AggressiveInlining)]
502 protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
503     ↪ second);
504
505 [MethodImpl(MethodImplOptions.AggressiveInlining)]
506 protected virtual TLink Subtract(TLink first, TLink second) =>
507     ↪ Arithmetic<TLink>.Subtract(first, second);
508
509 [MethodImpl(MethodImplOptions.AggressiveInlining)]
510 protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
511
512 [MethodImpl(MethodImplOptions.AggressiveInlining)]
513 protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
514
515 #region Disposable
516
517 protected override bool AllowMultipleDisposeCalls
518 {
519     [MethodImpl(MethodImplOptions.AggressiveInlining)]
520     get => true;
521 }
522
523 [MethodImpl(MethodImplOptions.AggressiveInlining)]
524 protected override void Dispose(bool manual, bool wasDisposed)
525 {
526     if (!wasDisposed)
527     {
528         ResetPointers();
529         _memory.DisposeIfPossible();
530     }
531 }
532
533 #endregion
534 }

```

## 1.66 ./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 =
        ↳ 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) =>
        ↳ GetLinkReference(element).Source = previous;
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected override void SetNext(TLink element, TLink next) =>
        ↳ GetLinkReference(element).Target = next;
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
59 }
60 }

```

## 1.67 ./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 =
            ↳ EqualityComparer<TLink>.Default;
13
14         public static readonly long SizeInBytes = Structure<RawLink<TLink>>.Size;
15
16         public TLink Source;
17         public TLink Target;

```

```

18     public TLink LeftAsSource;
19     public TLink RightAsSource;
20     public TLink SizeAsSource;
21     public TLink LeftAsTarget;
22     public TLink RightAsTarget;
23     public TLink SizeAsTarget;
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     public override bool Equals(object obj) => obj is RawLink<TLink> link ? Equals(link) :
        ↳ false;
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     public bool Equals(RawLink<TLink> other)
30     => _equalityComparer.Equals(Source, other.Source)
31     && _equalityComparer.Equals(Target, other.Target)
32     && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
33     && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
34     && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
35     && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
36     && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
37     && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     public override int GetHashCode() => (Source, Target, LeftAsSource, RightAsSource,
        ↳ SizeAsSource, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     public static bool operator ==(RawLink<TLink> left, RawLink<TLink> right) =>
        ↳ left.Equals(right);
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     public static bool operator !=(RawLink<TLink> left, RawLink<TLink> right) => !(left ==
        ↳ right);
47 }
48 }

```

## 1.68 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSizeBalancedTreeMethodsBase.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 UInt32LinksSizeBalancedTreeMethodsBase :
9         ↳ LinksSizeBalancedTreeMethodsBase<uint>
10     {
11         protected new readonly RawLink<uint>* Links;
12         protected new readonly LinksHeader<uint>* Header;
13
14         protected UInt32LinksSizeBalancedTreeMethodsBase(LinksConstants<uint> constants,
15             ↳ RawLink<uint>* links, LinksHeader<uint>* header)
16             : base(constants, (byte*)links, (byte*)header)
17         {
18             Links = links;
19             Header = header;
20
21             [MethodImpl(MethodImplOptions.AggressiveInlining)]
22             protected override uint GetZero() => 0U;
23
24             [MethodImpl(MethodImplOptions.AggressiveInlining)]
25             protected override bool EqualToZero(uint value) => value == 0U;
26
27             [MethodImpl(MethodImplOptions.AggressiveInlining)]
28             protected override bool AreEqual(uint first, uint second) => first == second;
29
30             [MethodImpl(MethodImplOptions.AggressiveInlining)]
31             protected override bool GreaterThanZero(uint value) => value > 0U;
32
33             [MethodImpl(MethodImplOptions.AggressiveInlining)]
34             protected override bool GreaterThan(uint first, uint second) => first > second;
35
36             [MethodImpl(MethodImplOptions.AggressiveInlining)]
37             protected override bool GreaterOrEqualThan(uint first, uint second) => first >= second;
38
39             [MethodImpl(MethodImplOptions.AggressiveInlining)]
40             protected override bool GreaterOrEqualThanZero(uint value) => true; // value >= 0 is
                ↳ always true for uint

```

```

41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 protected override bool LessOrEqualThanZero(uint value) => value == 0U; // value is
   ↳ always >= 0 for uint
43
44 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 protected override bool LessOrEqualThan(uint first, uint second) => first <= second;
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 protected override bool LessThanZero(uint value) => false; // value < 0 is always false
   ↳ for uint
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 protected override bool LessThan(uint first, uint second) => first < second;
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected override uint Increment(uint value) => ++value;
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 protected override uint Decrement(uint value) => --value;
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected override uint Add(uint first, uint second) => first + second;
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override uint Subtract(uint first, uint second) => first - second;
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override bool FirstIsToTheLeftOfSecond(uint first, uint 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(uint first, uint 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<uint> GetHeaderReference() => ref *Header;
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 protected override ref RawLink<uint> GetLinkReference(uint link) => ref Links[link];
86 }
87 }

```

## 1.69 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSourcesSizeBalancedTreeMethods.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 UInt32LinksSourcesSizeBalancedTreeMethods :
   ↳ UInt32LinksSizeBalancedTreeMethodsBase
8     {
9         public UInt32LinksSourcesSizeBalancedTreeMethods(LinksConstants<uint> constants,
   ↳ RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links, header) { }
10
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsSource;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref uint GetRightReference(uint node) => ref
   ↳ Links[node].RightAsSource;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected override uint GetLeft(uint node) => Links[node].LeftAsSource;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override uint GetRight(uint node) => Links[node].RightAsSource;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

24     protected override void SetLeft(uint node, uint left) => Links[node].LeftAsSource = left;
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected override void SetRight(uint node, uint right) => Links[node].RightAsSource =
        ↳ right;
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     protected override uint GetSize(uint node) => Links[node].SizeAsSource;
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     protected override void SetSize(uint node, uint size) => Links[node].SizeAsSource = size;
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override uint GetTreeRoot() => Header->RootAsSource;
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override uint GetBasePartValue(uint link) => Links[link].Source;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
        ↳ uint secondSource, uint secondTarget)
43         => firstSource < secondSource || (firstSource == secondSource && firstTarget <
        ↳ secondTarget);
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
        ↳ uint secondSource, uint secondTarget)
47         => firstSource > secondSource || (firstSource == secondSource && firstTarget >
        ↳ secondTarget);
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override void ClearNode(uint node)
51     {
52         ref var link = ref Links[node];
53         link.LeftAsSource = 0U;
54         link.RightAsSource = 0U;
55         link.SizeAsSource = 0U;
56     }
57 }
58 }

```

## 1.70 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksTargetsSizeBalancedTreeMethods.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 UInt32LinksTargetsSizeBalancedTreeMethods :
        ↳ UInt32LinksSizeBalancedTreeMethodsBase
8      {
9          public UInt32LinksTargetsSizeBalancedTreeMethods(LinksConstants<uint> constants,
        ↳ RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links, header) { }
10
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsTarget;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref uint GetRightReference(uint node) => ref
        ↳ Links[node].RightAsTarget;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected override uint GetLeft(uint node) => Links[node].LeftAsTarget;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override uint GetRight(uint node) => Links[node].RightAsTarget;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override void SetLeft(uint node, uint left) => Links[node].LeftAsTarget = left;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override void SetRight(uint node, uint right) => Links[node].RightAsTarget =
        ↳ right;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override uint GetSize(uint node) => Links[node].SizeAsTarget;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetSize(uint node, uint size) => Links[node].SizeAsTarget = size;

```

```

34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected override uint GetTreeRoot() => Header->RootAsTarget;
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override uint GetBasePartValue(uint link) => Links[link].Target;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
42         ↪ uint secondSource, uint secondTarget)
43         => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
44             ↪ secondSource);
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
48         ↪ uint secondSource, uint secondTarget)
49         => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
50             ↪ secondSource);
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected override void ClearNode(uint node)
54     {
55         ref var link = ref Links[node];
56         link.LeftAsTarget = 0U;
57         link.RightAsTarget = 0U;
58         link.SizeAsTarget = 0U;
59     }
60 }

```

## 1.71 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32UnitedMemoryLinks.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
13     ↪ organizing the storage of links with addresses represented as <see cref="uint" />.</para>
14     /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
15     ↪ размером, для организации хранения связей с адресами представленными в виде <see
16     ↪ cref="uint"/>.</para>
17     /// </summary>
18     public unsafe class UInt32UnitedMemoryLinks : UnitedMemoryLinksBase<uint>
19     {
20         private readonly Func<ILinksTreeMethods<uint>> _createSourceTreeMethods;
21         private readonly Func<ILinksTreeMethods<uint>> _createTargetTreeMethods;
22         private LinksHeader<uint>* _header;
23         private RawLink<uint>* _links;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public UInt32UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
27
28         /// <summary>
29         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
30         ↪ минимальным шагом расширения базы данных.
31         /// </summary>
32         /// <param name="address">Полный путь к файлу базы данных.</param>
33         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
34         ↪ байтах.</param>
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public UInt32UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
37             ↪ FileMappedResizableDirectMemory(address, memoryReservationStep),
38             ↪ memoryReservationStep) { }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public UInt32UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
42             ↪ DefaultLinksSizeStep) { }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
46             ↪ memoryReservationStep) : this(memory, memoryReservationStep,
47             ↪ Default<LinksConstants<uint>>.Instance, IndexTreeType.Default) { }
48     }
49 }

```



```

39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
    ↳ memoryReservationStep, LinksConstants<uint> constants, IndexTreeType indexTreeType)
    ↳ : base(memory, memoryReservationStep, constants)
41 {
42     _createSourceTreeMethods = () => new
    ↳ UInt32LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
43     _createTargetTreeMethods = () => new
    ↳ UInt32LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
44     Init(memory, memoryReservationStep);
45 }
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 protected override void SetPointers(IResizableDirectMemory memory)
49 {
50     _header = (LinksHeader<uint>*)memory.Pointer;
51     _links = (RawLink<uint>*)memory.Pointer;
52     SourcesTreeMethods = _createSourceTreeMethods();
53     TargetsTreeMethods = _createTargetTreeMethods();
54     UnusedLinksListMethods = new UInt32UnusedLinksListMethods(_links, _header);
55 }
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 protected override void ResetPointers()
59 {
60     base.ResetPointers();
61     _links = null;
62     _header = null;
63 }
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override ref LinksHeader<uint> GetHeaderReference() => ref *_header;
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 protected override ref RawLink<uint> GetLinkReference(uint linkIndex) => ref
    ↳ _links[linkIndex];
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 protected override bool AreEqual(uint first, uint second) => first == second;
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected override bool LessThan(uint first, uint second) => first < second;
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected override bool LessOrEqualThan(uint first, uint second) => first <= second;
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 protected override bool GreaterThan(uint first, uint second) => first > second;
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 protected override bool GreaterOrEqualThan(uint first, uint second) => first >= second;
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 protected override uint GetZero() => 0U;
88
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 protected override uint GetOne() => 1U;
91
92 [MethodImpl(MethodImplOptions.AggressiveInlining)]
93 protected override long ConvertToInt64(uint value) => (long)value;
94
95 [MethodImpl(MethodImplOptions.AggressiveInlining)]
96 protected override uint ConvertToAddress(long value) => (uint)value;
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 protected override uint Add(uint first, uint second) => first + second;
100
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 protected override uint Subtract(uint first, uint second) => first - second;
103
104 [MethodImpl(MethodImplOptions.AggressiveInlining)]
105 protected override uint Increment(uint link) => ++link;
106
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]
108 protected override uint Decrement(uint link) => --link;
109 }
110 }

```

## 1.72 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32UnusedLinksListMethods.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 UInt32UnusedLinksListMethods : UnusedLinksListMethods<uint>
9     {
10         private readonly RawLink<uint>* _links;
11         private readonly LinksHeader<uint>* _header;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public UInt32UnusedLinksListMethods(RawLink<uint>* links, LinksHeader<uint>* 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<uint> GetLinkReference(uint link) => ref _links[link];
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override ref LinksHeader<uint> GetHeaderReference() => ref *_header;
26     }
27 }

```

## 1.73 ./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() => 0UL;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override bool EqualToZero(ulong value) => value == 0UL;
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 > 0UL;
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 == 0UL; // 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 }

```

```

47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
49     ↪ for ulong
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override bool LessThan(ulong first, ulong second) => first < second;
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected override ulong Increment(ulong value) => ++value;
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 protected override ulong Decrement(ulong value) => --value;
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected override ulong Add(ulong first, ulong second) => first + second;
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected override ulong Subtract(ulong first, ulong second) => first - second;
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 protected override bool FirstIsToLeftOfSecond(ulong first, ulong second)
68 {
69     ref var firstLink = ref Links[first];
70     ref var secondLink = ref Links[second];
71     return FirstIsToLeftOfSecond(firstLink.Source, firstLink.Target,
72         ↪ secondLink.Source, secondLink.Target);
73 }
74
75 [MethodImpl(MethodImplOptions.AggressiveInlining)]
76 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
77 {
78     ref var firstLink = ref Links[first];
79     ref var secondLink = ref Links[second];
80     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
81         ↪ secondLink.Source, secondLink.Target);
82 }
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 protected override ulong GetSizeValue(ulong value) => (value & 4294967264UL) >> 5;
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
89     ↪ storedValue & 31UL | (size & 134217727UL) << 5;
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 protected override bool GetLeftIsChildValue(ulong value) => (value & 16UL) >> 4 == 1UL;
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
96     ↪ storedValue = storedValue & 4294967279UL | (As<bool, byte>(ref value) & 1UL) << 4;
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 protected override bool GetRightIsChildValue(ulong value) => (value & 8UL) >> 3 == 1UL;
100
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>
103     ↪ storedValue = storedValue & 4294967287UL | (As<bool, byte>(ref value) & 1UL) << 3;
104
105 [MethodImpl(MethodImplOptions.AggressiveInlining)]
106 protected override sbyte GetBalanceValue(ulong value) => unchecked((sbyte)(value & 7UL |
107     ↪ 0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of
108     ↪ sbyte
109
110 [MethodImpl(MethodImplOptions.AggressiveInlining)]
111 protected override void SetBalanceValue(ref ulong storedValue, sbyte value) =>
112     ↪ storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
113     ↪ value & 3) & 7UL);
114
115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
120
121 }
122

```

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

```

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

```

## 1.75 ./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 :
8          ↪ UInt64LinksAvlBalancedTreeMethodsBase
9      {
10         public UInt64LinksSourcesAvlBalancedTreeMethods(LinksConstants<ulong> constants,
11             ↪ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
12             ↪ { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetLeftReference(ulong node) => ref
16             ↪ Links[node].LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20             ↪ Links[node].RightAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
30             ↪ left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
34             ↪ right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsSource);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
41             ↪ Links[node].SizeAsSource, size);
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override bool GetLeftIsChild(ulong node) =>
45             ↪ GetLeftIsChildValue(Links[node].SizeAsSource);
46
47         // [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         // protected override bool GetLeftIsChild(ulong node) => IsChild(node, GetLeft(node));
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override void SetLeftIsChild(ulong node, bool value) =>
52             ↪ SetLeftIsChildValue(ref Links[node].SizeAsSource, value);
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         protected override bool GetRightIsChild(ulong node) =>
56             ↪ GetRightIsChildValue(Links[node].SizeAsSource);
57
58         // [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         // protected override bool GetRightIsChild(ulong node) => IsChild(node, GetRight(node));
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

## 1.76 ./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         public UInt64LinksSourcesSizeBalancedTreeMethods(LinksConstants<ulong> constants,
11             ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
12             ↳ { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetLeftReference(ulong node) => ref
16             ↳ Links[node].LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20             ↳ Links[node].RightAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
30             ↳ left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
34             ↳ right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override ulong GetSize(ulong node) => Links[node].SizeAsSource;
38     }
39 }

```

```

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

```

## 1.77 ./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 :
    ↪ UInt64LinksAvlBalancedTreeMethodsBase
8      {
9          public UInt64LinksTargetsAvlBalancedTreeMethods(LinksConstants<ulong> constants,
    ↪ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
    ↪ { }
10
11     [MethodImpl(MethodImplOptions.AggressiveInlining)]
12     protected override ref ulong GetLeftReference(ulong node) => ref
    ↪ Links[node].LeftAsTarget;
13
14     [MethodImpl(MethodImplOptions.AggressiveInlining)]
15     protected override ref ulong GetRightReference(ulong node) => ref
    ↪ Links[node].RightAsTarget;
16
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
    ↪ left;
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
    ↪ right;
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsTarget);
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
    ↪ Links[node].SizeAsTarget, size);
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override bool GetLeftIsChild(ulong node) =>
    ↪ GetLeftIsChildValue(Links[node].SizeAsTarget);

```

```

37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override void SetLeftIsChild(ulong node, bool value) =>
39         ↳ SetLeftIsChildValue(ref Links[node].SizeAsTarget, value);
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override bool GetRightIsChild(ulong node) =>
43         ↳ GetRightIsChildValue(Links[node].SizeAsTarget);
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override void SetRightIsChild(ulong node, bool value) =>
47         ↳ SetRightIsChildValue(ref Links[node].SizeAsTarget, value);
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override sbyte GetBalance(ulong node) =>
51         ↳ GetBalanceValue(Links[node].SizeAsTarget);
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override ulong GetTreeRoot() => Header->RootAsTarget;
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
61         ↳ ulong secondSource, ulong secondTarget)
62         => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
63             ↳ secondSource);
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
67         ↳ ulong secondSource, ulong secondTarget)
68         => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
69             ↳ secondSource);
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     protected override void ClearNode(ulong node)
73     {
74         ref var link = ref Links[node];
75         link.LeftAsTarget = OUL;
76         link.RightAsTarget = OUL;
77         link.SizeAsTarget = OUL;
78     }
79 }

```

## 1.78 ./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
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetLeftReference(ulong node) => ref
16             ↳ Links[node].LeftAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20             ↳ Links[node].RightAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
27
28     }
29 }

```



```

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

```

## 1.79 ./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, IndexTreeType.Default) { }
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
    ↳ memoryReservationStep, LinksConstants<ulong> constants, IndexTreeType indexTreeType)
    ↳ : base(memory, memoryReservationStep, constants)
41 {
42     if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
43     {
44         _createSourceTreeMethods = () => new
            ↳ UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
45         _createTargetTreeMethods = () => new
            ↳ UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
46     }
47     else
48     {
49         _createSourceTreeMethods = () => new
            ↳ UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
50         _createTargetTreeMethods = () => new
            ↳ UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
51     }
52     Init(memory, memoryReservationStep);
53 }
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 protected override void SetPointers(IResizableDirectMemory memory)
57 {
58     _header = (LinksHeader<ulong>*)memory.Pointer;
59     _links = (RawLink<ulong>*)memory.Pointer;
60     SourcesTreeMethods = _createSourceTreeMethods();
61     TargetsTreeMethods = _createTargetTreeMethods();
62     UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
63 }
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override void ResetPointers()
67 {
68     base.ResetPointers();
69     _links = null;
70     _header = null;
71 }
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
    ↳ _links[linkIndex];
78
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 protected override bool AreEqual(ulong first, ulong second) => first == second;
81
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 protected override bool LessThan(ulong first, ulong second) => first < second;
84
85 [MethodImpl(MethodImplOptions.AggressiveInlining)]
86 protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
87
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 protected override bool GreaterThan(ulong first, ulong second) => first > second;
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 protected override ulong GetZero() => OUL;

```

```

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

```

#### 1.80 ./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 }

```

#### 1.81 ./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)]

```

```

23     public TLink Convert(TLink number)
24     {
25         var links = _links;
26         var nullConstant = links.Constants.Null;
27         var target = nullConstant;
28         for (var i = 0; !_equalityComparer.Equals(number, _zero) && i <
29             ↪ NumericType<TLink>.BitsSize; i++)
30         {
31             if (_equalityComparer.Equals(Bit.And(number, _one), _one))
32             {
33                 target = _equalityComparer.Equals(target, nullConstant)
34                     ? _powerOf2ToUnaryNumberConverter.Convert(i)
35                     : links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
36             }
37             number = Bit.ShiftRight(number, 1);
38         }
39         return target;
40     }
41 }

```

## 1.82 ./csharp/Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConverter.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 LinkToItsFrequencyNumberConverter<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 LinkToItsFrequencyNumberConverter(
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 }

```

## 1.83 ./csharp/Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs

```

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

```

```

8
9 namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
12         ↳ IConverter<int, TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↳ EqualityComparer<TLink>.Default;
16
17         private readonly TLink[] _unaryNumberPowersOf2;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
21         {
22             _unaryNumberPowersOf2 = new TLink[64];
23             _unaryNumberPowersOf2[0] = one;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Convert(int power)
28         {
29             Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
30                 ↳ - 1), nameof(power));
31             if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
32             {
33                 return _unaryNumberPowersOf2[power];
34             }
35             var previousPowerOf2 = Convert(power - 1);
36             var powerOf2 = _links.GetOrCreate(previousPowerOf2, previousPowerOf2);
37             _unaryNumberPowersOf2[power] = powerOf2;
38             return powerOf2;
39         }
40     }
41 }

```

#### 1.84 ./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;

```

```

41     var source = links.GetSource(unaryNumber);
42     var target = links.GetTarget(unaryNumber);
43     if (_equalityComparer.Equals(source, target))
44     {
45         return _unaryToUInt64[unaryNumber];
46     }
47     else
48     {
49         var result = _unaryToUInt64[source];
50         TLink lastValue;
51         while (!_unaryToUInt64.TryGetValue(target, out lastValue))
52         {
53             source = links.GetSource(target);
54             result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
55             target = links.GetTarget(target);
56         }
57         result = Arithmetic<TLink>.Add(result, lastValue);
58         return result;
59     }
60 }
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 private static Dictionary<TLink, TLink> CreateUnaryToUInt64Dictionary(ILinks<TLink>
    ↪ links, TLink unaryOne)
64 {
65     var unaryToUInt64 = new Dictionary<TLink, TLink>
66     {
67         { unaryOne, _one }
68     };
69     var unary = unaryOne;
70     var number = _one;
71     for (var i = 1; i < 64; i++)
72     {
73         unary = links.GetOrCreate(unary, unary);
74         number = Double(number);
75         unaryToUInt64.Add(unary, number);
76     }
77     return unaryToUInt64;
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 private static TLink Double(TLink number) =>
    ↪ _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
82 }
83 }

```

## 1.85 ./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>,
    ↪ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↪ EqualityComparer<TLink>.Default;
14         private static readonly TLink _zero = default;
15         private static readonly TLink _one = Arithmetic.Increment(_zero);
16
17         private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int>,
    ↪ TLink> powerOf2ToUnaryNumberConverter) : base(links) => _unaryNumberPowerOf2Indicies
    ↪ = CreateUnaryNumberPowerOf2IndiciesDictionary(powerOf2ToUnaryNumberConverter);
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public TLink Convert(TLink sourceNumber)
24         {
25             var links = _links;
26             var nullConstant = links.Constants.Null;
27             var source = sourceNumber;
28             var target = nullConstant;
29             if (!_equalityComparer.Equals(source, nullConstant))

```

```

30     {
31         while (true)
32         {
33             if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
34             {
35                 SetBit(ref target, powerOf2Index);
36                 break;
37             }
38             else
39             {
40                 powerOf2Index = _unaryNumberPowerOf2Indicies[links.GetSource(source)];
41                 SetBit(ref target, powerOf2Index);
42                 source = links.GetTarget(source);
43             }
44         }
45     }
46     return target;
47 }
48
49 [MethodImpl(MethodImplOptions.AggressiveInlining)]
50 private static Dictionary<TLink, int>
    ↪ CreateUnaryNumberPowerOf2IndiciesDictionary(IConverter<int, TLink>
    ↪ powerOf2ToUnaryNumberConverter)
51 {
52     var unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
53     for (int i = 0; i < NumericType<TLink>.BitsSize; i++)
54     {
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 }

```

## 1.86 ./csharp/Platform.Data.Doublets/PropertyOperators/PropertiesOperator.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 PropertiesOperator<TLink> : LinksOperatorBase<TLink>, IProperties<TLink, TLink,
    ↪ TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↪ EqualityComparer<TLink>.Default;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public PropertiesOperator(ILinks<TLink> links) : base(links) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public TLink GetValue(TLink @object, TLink property)
18         {
19             var links = _links;
20             var objectProperty = links.SearchOrDefault(@object, property);
21             if (_equalityComparer.Equals(objectProperty, default))
22             {
23                 return default;
24             }
25             var constants = links.Constants;
26             var any = constants.Any;
27             var query = new Link<TLink>(any, objectProperty, any);
28             var valueLink = links.SingleOrDefault(query);
29             if (valueLink == null)
30             {
31                 return default;
32             }
33             return links.GetTarget(valueLink[constants.IndexPart]);
34         }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public void SetValue(TLink @object, TLink property, TLink value)
38         {

```

```

39         var links = _links;
40         var objectProperty = links.GetOrCreate(@object, property);
41         links.DeleteMany(links.AllIndices(links.Constants.Any, objectProperty));
42         links.GetOrCreate(objectProperty, value);
43     }
44 }
45 }

```

## 1.87 ./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 =
12             ↪ EqualityComparer<TLink>.Default;
13
14         private readonly TLink _propertyMarker;
15         private readonly TLink _propertyValueMarker;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
19             ↪ propertyValueMarker) : base(links)
20         {
21             _propertyMarker = propertyMarker;
22             _propertyValueMarker = propertyValueMarker;
23         }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public TLink Get(TLink link)
27         {
28             var property = _links.SearchOrDefault(link, _propertyMarker);
29             return GetValue(GetContainer(property));
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         private TLink GetContainer(TLink property)
34         {
35             var valueContainer = default(TLink);
36             if (_equalityComparer.Equals(property, default))
37             {
38                 return valueContainer;
39             }
40             var links = _links;
41             var constants = links.Constants;
42             var countinueConstant = constants.Continue;
43             var breakConstant = constants.Break;
44             var anyConstant = constants.Any;
45             var query = new Link<TLink>(anyConstant, property, anyConstant);
46             links.Each(candidate =>
47             {
48                 var candidateTarget = links.GetTarget(candidate);
49                 var valueTarget = links.GetTarget(candidateTarget);
50                 if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
51                 {
52                     valueContainer = links.GetIndex(candidate);
53                     return breakConstant;
54                 }
55                 return countinueConstant;
56             }, query);
57             return valueContainer;
58         }
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
62             ↪ ? default : _links.GetTarget(container);
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         public void Set(TLink link, TLink value)
66         {
67             var links = _links;
68             var property = links.GetOrCreate(link, _propertyMarker);
69             var container = GetContainer(property);
70             if (_equalityComparer.Equals(container, default))
71             {

```



```

69         links.GetOrCreate(property, value);
70     }
71     else
72     {
73         links.Update(container, property, value);
74     }
75 }
76 }
77 }

```

## 1.88 ./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(ICollection<TLink> destination, ICollection<TLink> source, int length)
45             {
46                 var loopedLength = length - (length % 2);
47                 for (var i = 0; i < loopedLength; i += 2)
48                 {
49                     destination[i / 2] = _links.GetOrCreate(source[i], source[i + 1]);
50                 }
51                 if (length > loopedLength)
52                 {
53                     destination[length / 2] = source[length - 1];
54                 }
55             }
56         }
57     }

```

## 1.89 ./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         private struct HalfDoublet
41         {
42             public TLink Element;
43             public LinkFrequency<TLink> DoubletData;
44
45             [MethodImpl(MethodImplOptions.AggressiveInlining)]
46             public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
47             {
48                 Element = element;
49                 DoubletData = doubletData;
50             }
51
52             public override string ToString() => $"{Element}: ({DoubletData})";
53         }
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
57             ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
58             : this(links, baseConverter, doubletFrequenciesCache, _one, true) { }
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
62             ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
63             ↳ doInitialFrequenciesIncrement)
64             : this(links, baseConverter, doubletFrequenciesCache, _one,
65                 ↳ doInitialFrequenciesIncrement) { }
66
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
69             ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
70             ↳ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
71             : base(links)
72         {
73             _baseConverter = baseConverter;
74             _doubletFrequenciesCache = doubletFrequenciesCache;
75             if (_comparer.Compare(minFrequencyToCompress, _one) < 0)
76             {
77                 minFrequencyToCompress = _one;
78             }
79             _minFrequencyToCompress = minFrequencyToCompress;
80             _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
81             ResetMaxDoublet();
82         }
83
84         [MethodImpl(MethodImplOptions.AggressiveInlining)]
85         public override TLink Convert(IList<TLink> source) =>
86             ↳ _baseConverter.Convert(Compress(source));
87
88         /// <remarks>

```

```

77  /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
78  /// Faster version (doublets' frequencies dictionary is not recreated).
79  /// </remarks>
80  [MethodImpl(MethodImplOptions.AggressiveInlining)]
81  private IList<TLink> Compress(IList<TLink> sequence)
82  {
83      if (sequence.IsNullOrEmpty())
84      {
85          return null;
86      }
87      if (sequence.Count == 1)
88      {
89          return sequence;
90      }
91      if (sequence.Count == 2)
92      {
93          return new[] { _links.GetOrCreate(sequence[0], sequence[1]) };
94      }
95      // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
96      var copy = new HalfDoublet[sequence.Count];
97      Doublet<TLink> doublet = default;
98      for (var i = 1; i < sequence.Count; i++)
99      {
100         doublet = new Doublet<TLink>(sequence[i - 1], sequence[i]);
101         LinkFrequency<TLink> data;
102         if (_doInitialFrequenciesIncrement)
103         {
104             data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
105         }
106         else
107         {
108             data = _doubletFrequenciesCache.GetFrequency(ref doublet);
109             if (data == null)
110             {
111                 throw new NotSupportedException("If you ask not to increment
112                     ↪ frequencies, it is expected that all frequencies for the sequence
113                     ↪ are prepared.");
114             }
115             copy[i - 1].Element = sequence[i - 1];
116             copy[i - 1].DoubletData = data;
117             UpdateMaxDoublet(ref doublet, data);
118         }
119         copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
120         copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
121         if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
122         {
123             var newLength = ReplaceDoublets(copy);
124             sequence = new TLink[newLength];
125             for (int i = 0; i < newLength; i++)
126             {
127                 sequence[i] = copy[i].Element;
128             }
129         }
130         return sequence;
131     }
132     /// <remarks>
133     /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
134     /// </remarks>
135     [MethodImpl(MethodImplOptions.AggressiveInlining)]
136     private int ReplaceDoublets(HalfDoublet[] copy)
137     {
138         var oldLength = copy.Length;
139         var newLength = copy.Length;
140         while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
141         {
142             var maxDoubletSource = _maxDoublet.Source;
143             var maxDoubletTarget = _maxDoublet.Target;
144             if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
145             {
146                 _maxDoubletData.Link = _links.GetOrCreate(maxDoubletSource,
147                     ↪ maxDoubletTarget);
148             }
149             var maxDoubletReplacementLink = _maxDoubletData.Link;
150             oldLength--;
151             var oldLengthMinusTwo = oldLength - 1;
152             // Substitute all usages
153             int w = 0, r = 0; // (r == read, w == write)

```

```

153     for (; r < oldLength; r++)
154     {
155         if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
156             ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
157         {
158             if (r > 0)
159             {
160                 var previous = copy[w - 1].Element;
161                 copy[w - 1].DoubletData.DecrementFrequency();
162                 copy[w - 1].DoubletData =
163                     ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
164                     ↪ maxDoubletReplacementLink);
165             }
166             if (r < oldLengthMinusTwo)
167             {
168                 var next = copy[r + 2].Element;
169                 copy[r + 1].DoubletData.DecrementFrequency();
170                 copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
171                 ↪ next);
172             }
173             copy[w++] = copy[r];
174             newLength--;
175         }
176         else
177         {
178             copy[w++] = copy[r];
179         }
180     }
181     if (w < newLength)
182     {
183         copy[w] = copy[r];
184     }
185     oldLength = newLength;
186     ResetMaxDoublet();
187     UpdateMaxDoublet(copy, newLength);
188 }
189 return newLength;
190 }
191
192 [MethodImpl(MethodImplOptions.AggressiveInlining)]
193 private void ResetMaxDoublet()
194 {
195     _maxDoublet = new Doublet<TLink>();
196     _maxDoubletData = new LinkFrequency<TLink>();
197 }
198
199 [MethodImpl(MethodImplOptions.AggressiveInlining)]
200 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
201 {
202     Doublet<TLink> doublet = default;
203     for (var i = 1; i < length; i++)
204     {
205         doublet = new Doublet<TLink>(copy[i - 1].Element, copy[i].Element);
206         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
207     }
208 }
209
210 [MethodImpl(MethodImplOptions.AggressiveInlining)]
211 private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
212 {
213     var frequency = data.Frequency;
214     var maxFrequency = _maxDoubletData.Frequency;
215     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
216     ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
217     ↪ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
218     ↪ _maxDoublet.Target)))
219     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
220         (_comparer.Compare(maxFrequency, frequency) < 0 ||
221         ↪ (_equalityComparer.Equals(maxFrequency, frequency) &&
222         ↪ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
223         ↪ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
224         ↪ better stability and better compression on sequent data and even on random
225         ↪ numbers data (but gives collisions anyway) */
226     {
227         _maxDoublet = doublet;
228         _maxDoubletData = data;
229     }
230 }

```

```

218     }
219 }
220 }
221 }

```

#### 1.90 ./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 }

```

#### 1.91 ./csharp/Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Collections.Lists;
4 using Platform.Converters;
5 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
6 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Sequences.Converters
11 {
12     public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ⇨ EqualityComparer<TLink>.Default;
16         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
17
18         private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
22             ⇨ sequenceToItsLocalElementLevelsConverter) : base(links)
23         => _sequenceToItsLocalElementLevelsConverter =
24             ⇨ sequenceToItsLocalElementLevelsConverter;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public OptimalVariantConverter(ILinks<TLink> links, LinkFrequenciesCache<TLink>
28             ⇨ linkFrequenciesCache)
29         : this(links, new SequenceToItsLocalElementLevelsConverter<TLink>(links, new Frequen
30             ⇨ ciesCacheBasedLinkToItsFrequencyNumberConverter<TLink>(linkFrequenciesCache))) { }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public OptimalVariantConverter(ILinks<TLink> links)
34         : this(links, new LinkFrequenciesCache<TLink>(links, new
35             ⇨ TotalSequenceSymbolFrequencyCounter<TLink>(links))) { }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public override TLink Convert(IList<TLink> sequence)
39         {
40             var length = sequence.Count;
41             if (length == 1)
42             {
43                 return sequence[0];
44             }
45             if (length == 2)
46             {
47                 return _links.GetOrCreate(sequence[0], sequence[1]);
48             }
49             sequence = sequence.ToArray();
50             var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
51             while (length > 2)
52             {

```

```

47     var levelRepeat = 1;
48     var currentLevel = levels[0];
49     var previousLevel = levels[0];
50     var skipOnce = false;
51     var w = 0;
52     for (var i = 1; i < length; i++)
53     {
54         if (_equalityComparer.Equals(currentLevel, levels[i]))
55         {
56             levelRepeat++;
57             skipOnce = false;
58             if (levelRepeat == 2)
59             {
60                 sequence[w] = _links.GetOrCreate(sequence[i - 1], sequence[i]);
61                 var newLevel = i >= length - 1 ?
62                     GetPreviousLowerThanCurrentOrCurrent(previousLevel,
63                         ↪ currentLevel) :
64                     i < 2 ?
65                         GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
66                         GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
67                             ↪ currentLevel, levels[i + 1]);
68                 levels[w] = newLevel;
69                 previousLevel = currentLevel;
70                 w++;
71                 levelRepeat = 0;
72                 skipOnce = true;
73             }
74             else if (i == length - 1)
75             {
76                 sequence[w] = sequence[i];
77                 levels[w] = levels[i];
78                 w++;
79             }
80         }
81         else
82         {
83             currentLevel = levels[i];
84             levelRepeat = 1;
85             if (skipOnce)
86             {
87                 skipOnce = false;
88             }
89             else
90             {
91                 sequence[w] = sequence[i - 1];
92                 levels[w] = levels[i - 1];
93                 previousLevel = levels[w];
94                 w++;
95             }
96             if (i == length - 1)
97             {
98                 sequence[w] = sequence[i];
99                 levels[w] = levels[i];
100                 w++;
101             }
102         }
103     }
104     length = w;
105     return _links.GetOrCreate(sequence[0], sequence[1]);
106 }
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]
108 private static TLink GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink previous, TLink
109     ↪ current, TLink next)
110 {
111     return _comparer.Compare(previous, next) > 0
112         ? _comparer.Compare(previous, current) < 0 ? previous : current
113         : _comparer.Compare(next, current) < 0 ? next : current;
114 }
115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
117     ↪ _comparer.Compare(next, current) < 0 ? next : current;
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
120     ↪ => _comparer.Compare(previous, current) < 0 ? previous : current;

```

121 }

## 1.92 ./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>,
10     ↪ IConverter<IList<TLink>>
11     {
12         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
13
14         private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
18         ↪ IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
19         ↪ => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public IList<TLink> Convert(IList<TLink> sequence)
23         {
24             var levels = new TLink[sequence.Count];
25             levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
26             for (var i = 1; i < sequence.Count - 1; i++)
27             {
28                 var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
29                 var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
30                 levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
31             }
32             levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
33             ↪ sequence[sequence.Count - 1]);
34             return levels;
35         }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public TLink GetFrequencyNumber(TLink source, TLink target) =>
39         ↪ _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
40     }
41 }
```

## 1.93 ./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs

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

## 1.94 ./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Interfaces;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Sequences.CriterionMatchers
8 {
9     public class MarkedSequenceCriterionMatcher<TLink> : ICriterionMatcher<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12         ↪ EqualityComparer<TLink>.Default;
13
14         private readonly ILinks<TLink> _links;
```

```

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,
27             ↪ sequenceCandidate), _links.Constants.Null);
28 }

```

## 1.95 ./csharp/Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Collections.Stacks;
4  using Platform.Data.Doublets.Sequences.HeightProviders;
5  using Platform.Data.Sequences;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Sequences
10 {
11     public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
12         ↪ ISequenceAppender<TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↪ EqualityComparer<TLink>.Default;
16
17         private readonly IStack<TLink> _stack;
18         private readonly ISequenceHeightProvider<TLink> _heightProvider;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
22             ↪ ISequenceHeightProvider<TLink> heightProvider)
23             : base(links)
24         {
25             _stack = stack;
26             _heightProvider = heightProvider;
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public TLink Append(TLink sequence, TLink appendant)
31         {
32             var cursor = sequence;
33             var links = _links;
34             while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
35             {
36                 var source = links.GetSource(cursor);
37                 var target = links.GetTarget(cursor);
38                 if (_equalityComparer.Equals(_heightProvider.Get(source),
39                     ↪ _heightProvider.Get(target)))
40                 {
41                     break;
42                 }
43                 else
44                 {
45                     _stack.Push(source);
46                     cursor = target;
47                 }
48             }
49             var left = cursor;
50             var right = appendant;
51             while (!_equalityComparer.Equals(cursor = _stack.Pop(), links.Constants.Null))
52             {
53                 right = links.GetOrCreate(left, right);
54                 left = cursor;
55             }
56             return links.GetOrCreate(left, right);
57         }
58     }
59 }

```



## 1.96 ./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 }

```

## 1.97 ./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs

```

1 using System;
2 using System.Linq;
3 using System.Collections.Generic;
4 using System.Runtime.CompilerServices;
5 using Platform.Interfaces;
6 using Platform.Collections;
7 using Platform.Collections.Lists;
8 using Platform.Collections.Segments;
9 using Platform.Collections.Segments.Walkers;
10 using Platform.Singletons;
11 using Platform.Converters;
12 using Platform.Data.Doublets.Unicode;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     public class DuplicateSegmentsProvider<TLink> :
19         ↪ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
20         ↪ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
21     {
22         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
23             ↪ UncheckedConverter<TLink, long>.Default;
24         private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
25             ↪ UncheckedConverter<TLink, ulong>.Default;
26         private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
27             ↪ UncheckedConverter<ulong, TLink>.Default;
28
29         private readonly ILinks<TLink> _links;
30         private readonly ILinks<TLink> _sequences;
31         private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
32         private BitString _visited;
33
34         private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
35             ↪ IList<TLink>>>
36         {
37             private readonly IListEqualityComparer<TLink> _listComparer;
38
39             public ItemEquilityComparer() => _listComparer =
40                 ↪ Default<IListEqualityComparer<TLink>>.Instance;
41
42             [MethodImpl(MethodImplOptions.AggressiveInlining)]
43             public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
44                 ↪ KeyValuePair<IList<TLink>, IList<TLink>> right) =>
45                 ↪ _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
46                 ↪ right.Value);
47
48             [MethodImpl(MethodImplOptions.AggressiveInlining)]
49             public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
50                 ↪ (_listComparer.GetHashCode(pair.Key),
51                 ↪ _listComparer.GetHashCode(pair.Value)).GetHashCode();
52         }
53
54         private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
55         {
56
57         }
58     }
59 }

```

```

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,
51         ↪ KeyValuePair<IList<TLink>, IList<TLink>> right)
52     {
53         var intermediateResult = _listComparer.Compare(left.Key, right.Key);
54         if (intermediateResult == 0)
55         {
56             intermediateResult = _listComparer.Compare(left.Value, right.Value);
57         }
58         return intermediateResult;
59     }
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
63         : base(minimumStringSegmentLength: 2)
64     {
65         _links = links;
66         _sequences = sequences;
67     }
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
71     {
72         _groups = new HashSet<KeyValuePair<IList<TLink>,
73             ↪ IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
74         var links = _links;
75         var count = links.Count();
76         _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
77         links.Each(link =>
78         {
79             var linkIndex = links.GetIndex(link);
80             var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
81             var constants = links.Constants;
82             if (!_visited.Get(linkBitIndex))
83             {
84                 var sequenceElements = new List<TLink>();
85                 var filler = new ListFiller<TLink, TLink>(sequenceElements, constants.Break);
86                 _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
87                     ↪ LinkAddress<TLink>(linkIndex));
88                 if (sequenceElements.Count > 2)
89                 {
90                     WalkAll(sequenceElements);
91                 }
92             }
93             return constants.Continue;
94         });
95         var resultList = _groups.ToList();
96         var comparer = Default<ItemComparer>.Instance;
97         resultList.Sort(comparer);
98
99         #if DEBUG
100         foreach (var item in resultList)
101         {
102             PrintDuplications(item);
103         }
104
105         #endif
106         return resultList;
107     }
108
109     [MethodImpl(MethodImplOptions.AggressiveInlining)]
110     protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
111         ↪ length) => new Segment<TLink>(elements, offset, length);
112
113     [MethodImpl(MethodImplOptions.AggressiveInlining)]
114     protected override void OnDuplicateFound(Segment<TLink> segment)
115     {
116         var duplicates = CollectDuplicationsForSegment(segment);
117         if (duplicates.Count > 1)
118         {
119             _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
120                 ↪ duplicates));
121         }
122     }

```

```

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_
145             ↪ ashSet<ulong>)(object)readAsElement,
146             ↪ (IList<ulong>)segment);
147         foreach (var partiallyMatchedSequence in partiallyMatched)
148         {
149             var sequenceIndex =
150                 ↪ _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
151             duplicates.Add(sequenceIndex);
152         }
153     }
154     duplicates.Sort();
155     return duplicates;
156 }
157
158 [MethodImpl(MethodImplOptions.AggressiveInlining)]
159 private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
160 {
161     if (!(_links is ILinks<ulong> ulongLinks))
162     {
163         return;
164     }
165     var duplicatesKey = duplicatesItem.Key;
166     var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
167     Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)}");
168     var duplicatesList = duplicatesItem.Value;
169     for (int i = 0; i < duplicatesList.Count; i++)
170     {
171         var sequenceIndex = _addressToUInt64Converter.Convert(duplicatesList[i]);
172         var formattedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
173             ↪ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
174             ↪ UnicodeMap.IsCharLink(link.Index) ?
175             ↪ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
176         Console.WriteLine(formattedSequenceStructure);
177         var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
178             ↪ ulongLinks);
179         Console.WriteLine(sequenceString);
180     }
181     Console.WriteLine();
182 }
183 }
184 }

```

## 1.98 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs

```

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

```

```

10 {
11     /// <remarks>
12     /// Can be used to operate with many CompressingConverters (to keep global frequencies data
13     /// ↳ between them).
14     /// TODO: Extract interface to implement frequencies storage inside Links storage
15     /// </remarks>
16     public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
17     {
18         private static readonly EqualityComparer<TLink> _equalityComparer =
19             ↳ EqualityComparer<TLink>.Default;
20         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
21
22         private static readonly TLink _zero = default;
23         private static readonly TLink _one = Arithmetic.Increment(_zero);
24
25         private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
26         private readonly ICounter<TLink, TLink> _frequencyCounter;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
30             : base(links)
31         {
32             _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
33                 ↳ DoubletComparer<TLink>.Default);
34             _frequencyCounter = frequencyCounter;
35         }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
39         {
40             var doublet = new Doublet<TLink>(source, target);
41             return GetFrequency(ref doublet);
42         }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
46         {
47             _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
48             return data;
49         }
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         public void IncrementFrequencies(IList<TLink> sequence)
53         {
54             for (var i = 1; i < sequence.Count; i++)
55             {
56                 IncrementFrequency(sequence[i - 1], sequence[i]);
57             }
58         }
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
62         {
63             var doublet = new Doublet<TLink>(source, target);
64             return IncrementFrequency(ref doublet);
65         }
66
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         public void PrintFrequencies(IList<TLink> sequence)
69         {
70             for (var i = 1; i < sequence.Count; i++)
71             {
72                 PrintFrequency(sequence[i - 1], sequence[i]);
73             }
74         }
75
76         [MethodImpl(MethodImplOptions.AggressiveInlining)]
77         public void PrintFrequency(TLink source, TLink target)
78         {
79             var number = GetFrequency(source, target).Frequency;
80             Console.WriteLine("{0},{1}) - {2}", source, target, number);
81         }
82
83         [MethodImpl(MethodImplOptions.AggressiveInlining)]
84         public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
85         {
86             if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
87             {
88                 data.IncrementFrequency();
89             }
90         }
91     }
92 }

```

```

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             //{
121             //    if (value.Frequency > 0)
122             //    {
123             //        var frequency = value.Frequency;
124             //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
125             //        var count = _countLinkFrequency(linkIndex);
126             //        if ((frequency > count && frequency - count > 1) || (count > frequency
127             //            ↪ && count - frequency > 1))
128             //            throw new InvalidOperationException("Frequencies validation
129             //            ↪ failed.");
130             //    }
131             //}
132         }
133     }
134 }

```

## 1.99 ./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     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     public override string ToString() => $"F: {Frequency}, L: {Link}";
30 }
31 }
32 }

```

#### 1.100 ./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 }

```

#### 1.101 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs

```

1 using System.Runtime.CompilerServices;
2 using Platform.Interfaces;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7 {
8     public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
9         ↳ SequenceSymbolFrequencyOneOffCounter<TLink>
10     {
11         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
15             ↳ ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
16             : base(links, sequenceLink, symbol)
17             => _markedSequenceMatcher = markedSequenceMatcher;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public override TLink Count()
21         {
22             if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
23             {
24                 return default;
25             }
26             return base.Count();
27         }
28     }
29 }

```

#### 1.102 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Interfaces;
4 using Platform.Numbers;
5 using Platform.Data.Sequences;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
10 {
11     public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↳ EqualityComparer<TLink>.Default;
15         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
16     }
17 }

```

```

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,
23         ↪ TLink symbol)
24     {
25         _links = links;
26         _sequenceLink = sequenceLink;
27         _symbol = symbol;
28         _total = default;
29     }
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     public virtual TLink Count()
33     {
34         if (_comparer.Compare(_total, default) > 0)
35         {
36             return _total;
37         }
38         StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
39             ↪ IsElement, VisitElement);
40         return _total;
41     }
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
45         ↪ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
46         ↪ IsPartialPoint
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     private bool VisitElement(TLink element)
50     {
51         if (_equalityComparer.Equals(element, _symbol))
52         {
53             _total = Arithmetic.Increment(_total);
54         }
55         return true;
56     }
57 }

```

### 1.103 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequency

```

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 }

```

### 1.104 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequency

```

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 }

```

#### 1.105 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter

```

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

```

#### 1.106 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOff

```

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             {

```



```

34         return _total;
35     }
36     CountCore(_symbol);
37     return _total;
38 }
39
40 [MethodImpl(MethodImplOptions.AggressiveInlining)]
41 private void CountCore(TLink link)
42 {
43     var any = _links.Constants.Any;
44     if (_equalityComparer.Equals(_links.Count(any, link), default))
45     {
46         CountSequenceSymbolFrequency(link);
47     }
48     else
49     {
50         _links.Each(EachElementHandler, any, link);
51     }
52 }
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected virtual void CountSequenceSymbolFrequency(TLink link)
56 {
57     var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
58     ↪ link, _symbol);
59     _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
60 }
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 private TLink EachElementHandler(IList<TLink> doublet)
64 {
65     var constants = _links.Constants;
66     var doubletIndex = doublet[constants.IndexPart];
67     if (_visits.Add(doubletIndex))
68     {
69         CountCore(doubletIndex);
70     }
71     return constants.Continue;
72 }
73 }

```

#### 1.107 ./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         {

```

```

38     TLink height;
39     var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
40     if (_equalityComparer.Equals(heightValue, default))
41     {
42         height = _baseHeightProvider.Get(sequence);
43         heightValue = _addressToUnaryNumberConverter.Convert(height);
44         _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
45     }
46     else
47     {
48         height = _unaryNumberToAddressConverter.Convert(heightValue);
49     }
50     return height;
51 }
52 }
53 }

```

#### 1.108 ./csharp/Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs

```

1  using System.Runtime.CompilerServices;
2  using Platform.Interfaces;
3  using Platform.Numbers;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences.HeightProviders
8  {
9      public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
10         ↳ ISequenceHeightProvider<TLink>
11     {
12         private readonly ICriterionMatcher<TLink> _elementMatcher;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
16             ↳ elementMatcher) : base(links) => _elementMatcher = elementMatcher;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public TLink Get(TLink sequence)
20         {
21             var height = default(TLink);
22             var pairOrElement = sequence;
23             while (!_elementMatcher.IsMatched(pairOrElement))
24             {
25                 pairOrElement = _links.GetTarget(pairOrElement);
26                 height = Arithmetic.Increment(height);
27             }
28             return height;
29         }
30     }
31 }

```

#### 1.109 ./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 }

```

#### 1.110 ./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)]

```

```

16     public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
17         ↪ _cache = cache;
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     public bool Add(IList<TLink> sequence)
21     {
22         var indexed = true;
23         var i = sequence.Count;
24         while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
25             ↪ { }
26         for (; i >= 1; i--)
27         {
28             _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
29         }
30         return indexed;
31     }
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     private bool IsIndexedWithIncrement(TLink source, TLink target)
35     {
36         var frequency = _cache.GetFrequency(source, target);
37         if (frequency == null)
38         {
39             return false;
40         }
41         var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
42         if (indexed)
43         {
44             _cache.IncrementFrequency(source, target);
45         }
46         return indexed;
47     }
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     public bool MightContain(IList<TLink> sequence)
51     {
52         var indexed = true;
53         var i = sequence.Count;
54         while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
55         return indexed;
56     }
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     private bool IsIndexed(TLink source, TLink target)
60     {
61         var frequency = _cache.GetFrequency(source, target);
62         if (frequency == null)
63         {
64             return false;
65         }
66         return !_equalityComparer.Equals(frequency.Frequency, default);
67     }
68 }

```

#### 1.111 ./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.Incremeters;
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         }
26     }
27 }

```

```

22     _frequencyIncrementer = frequencyIncrementer;
23 }
24
25 [MethodImpl(MethodImplOptions.AggressiveInlining)]
26 public override bool Add(ICollection<TLink> sequence)
27 {
28     var indexed = true;
29     var i = sequence.Count;
30     while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
31         { }
32     for (; i >= 1; i--)
33     {
34         Increment(_links.GetOrCreate(sequence[i - 1], sequence[i]));
35     }
36     return indexed;
37 }
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 private bool IsIndexedWithIncrement(TLink source, TLink target)
41 {
42     var link = _links.SearchOrDefault(source, target);
43     var indexed = !_equalityComparer.Equals(link, default);
44     if (indexed)
45     {
46         Increment(link);
47     }
48     return indexed;
49 }
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 private void Increment(TLink link)
53 {
54     var previousFrequency = _frequencyPropertyOperator.Get(link);
55     var frequency = _frequencyIncrementer.Increment(previousFrequency);
56     _frequencyPropertyOperator.Set(link, frequency);
57 }
58 }

```

#### 1.112 ./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(ICollection<TLink> sequence);
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         bool MightContain(ICollection<TLink> sequence);
20     }
21 }

```

#### 1.113 ./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(ICollection<TLink> links) : base(links) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

16 public virtual bool Add(IList<TLink> sequence)
17 {
18     var indexed = true;
19     var i = sequence.Count;
20     while (--i >= 1 && (indexed =
        ↳ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
        ↳ default))) { }
21     for (; i >= 1; i--)
22     {
23         _links.GetOrCreate(sequence[i - 1], sequence[i]);
24     }
25     return indexed;
26 }
27
28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 public virtual bool MightContain(IList<TLink> sequence)
30 {
31     var indexed = true;
32     var i = sequence.Count;
33     while (--i >= 1 && (indexed =
        ↳ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
        ↳ default))) { }
34     return indexed;
35 }
36 }
37 }

```

#### 1.114 ./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 =
            ↳ EqualityComparer<TLink>.Default;
11
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(IList<TLink> sequence)
19         {
20             var indexed = true;
21             var i = sequence.Count;
22             var links = _links.Unsync;
23             _links.SyncRoot.ExecuteReadOperation(() =>
24             {
25                 while (--i >= 1 && (indexed =
                    ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                    ↳ sequence[i]), default))) { }
26             });
27             if (!indexed)
28             {
29                 _links.SyncRoot.ExecuteWriteOperation(() =>
30                 {
31                     for (; i >= 1; i--)
32                     {
33                         links.GetOrCreate(sequence[i - 1], sequence[i]);
34                     }
35                 });
36             }
37             return indexed;
38         }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public bool MightContain(IList<TLink> sequence)
42         {
43             var links = _links.Unsync;
44             return _links.SyncRoot.ExecuteReadOperation(() =>
45             {
46                 var indexed = true;
47                 var i = sequence.Count;

```

```

48         while (--i >= 1 && (indexed =
           ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
           ↳ sequence[i]), default))) { }
49         return indexed;
50     });
51 }
52 }
53 }

```

#### 1.115 ./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 }

```

#### 1.116 ./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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 public List<ulong> CreateAllVariants1(params ulong[] sequence)
88 {
89     return _sync.ExecuteWriteOperation(() =>
90     {
91         if (sequence.IsNullOrEmpty())
92         {
93             return new List<ulong>();
94         }
95         Links.Unsync.EnsureLinkExists(sequence);
96         if (sequence.Length == 1)
97         {
98             return new List<ulong> { sequence[0] };
99         }
100         var results = new
101             ↪ List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
102         return CreateAllVariants1Core(sequence, results);
103     });
104 }
105 [MethodImpl(MethodImplOptions.AggressiveInlining)]
106 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
107 {
108     if (sequence.Length == 2)
109     {
110         var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
111         if (link == Constants.Null)
112         {
113             throw new NotImplementedException("Creation cancellation is not
114             ↪ implemented.");
115         }
116         results.Add(link);
117         return results;
118     }
119     var innerSequenceLength = sequence.Length - 1;
120     var innerSequence = new ulong[innerSequenceLength];
121     for (var li = 0; li < innerSequenceLength; li++)
122     {
123         var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
124         if (link == Constants.Null)
125         {
126             throw new NotImplementedException("Creation cancellation is not
127             ↪ implemented.");
128         }
129     }
130 }

```

```

125     }
126     for (var isi = 0; isi < li; isi++)
127     {
128         innerSequence[isi] = sequence[isi];
129     }
130     innerSequence[li] = link;
131     for (var isi = li + 1; isi < innerSequenceLength; isi++)
132     {
133         innerSequence[isi] = sequence[isi + 1];
134     }
135     CreateAllVariants1Core(innerSequence, results);
136 }
137 return results;
138 }
139
140 #endregion
141
142 [MethodImpl(MethodImplOptions.AggressiveInlining)]
143 public HashSet<ulong> Each1(params ulong[] sequence)
144 {
145     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
146     Each1(link =>
147     {
148         if (!visitedLinks.Contains(link))
149         {
150             visitedLinks.Add(link); // изучить почему случаются повторы
151         }
152         return true;
153     }, sequence);
154     return visitedLinks;
155 }
156
157 [MethodImpl(MethodImplOptions.AggressiveInlining)]
158 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
159 {
160     if (sequence.Length == 2)
161     {
162         Links.Unsync.Each(sequence[0], sequence[1], handler);
163     }
164     else
165     {
166         var innerSequenceLength = sequence.Length - 1;
167         for (var li = 0; li < innerSequenceLength; li++)
168         {
169             var left = sequence[li];
170             var right = sequence[li + 1];
171             if (left == 0 && right == 0)
172             {
173                 continue;
174             }
175             var linkIndex = li;
176             ulong[] innerSequence = null;
177             Links.Unsync.Each(doublet =>
178             {
179                 if (innerSequence == null)
180                 {
181                     innerSequence = new ulong[innerSequenceLength];
182                     for (var isi = 0; isi < linkIndex; isi++)
183                     {
184                         innerSequence[isi] = sequence[isi];
185                     }
186                     for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
187                     {
188                         innerSequence[isi] = sequence[isi + 1];
189                     }
190                 }
191                 innerSequence[linkIndex] = doublet[Constants.IndexPart];
192                 Each1(handler, innerSequence);
193                 return Constants.Continue;
194             }, Constants.Any, left, right);
195         }
196     }
197 }
198
199 [MethodImpl(MethodImplOptions.AggressiveInlining)]
200 public HashSet<ulong> EachPart(params ulong[] sequence)
201 {
202     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
203     EachPartCore(link =>

```



```

204     {
205         var linkIndex = link[Constants.IndexPart];
206         if (!visitedLinks.Contains(linkIndex))
207         {
208             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
209         }
210         return Constants.Continue;
211     }, sequence);
212     return visitedLinks;
213 }
214
215 [MethodImpl(MethodImplOptions.AggressiveInlining)]
216 public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
217 {
218     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
219     EachPartCore(link =>
220     {
221         var linkIndex = link[Constants.IndexPart];
222         if (!visitedLinks.Contains(linkIndex))
223         {
224             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
225             return handler(new LinkAddress<LinkIndex>(linkIndex));
226         }
227         return Constants.Continue;
228     }, sequence);
229 }
230
231 [MethodImpl(MethodImplOptions.AggressiveInlining)]
232 private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
233     ↪ sequence)
234 {
235     if (sequence.IsNullOrEmpty())
236     {
237         return;
238     }
239     Links.EnsureLinkIsAnyOrExists(sequence);
240     if (sequence.Length == 1)
241     {
242         var link = sequence[0];
243         if (link > 0)
244         {
245             handler(new LinkAddress<LinkIndex>(link));
246         }
247         else
248         {
249             Links.Each(Constants.Any, Constants.Any, handler);
250         }
251     }
252     else if (sequence.Length == 2)
253     {
254         //_links.Each(sequence[0], sequence[1], handler);
255         //  o_|      x_o ...
256         // x_|      |__|
257         Links.Each(sequence[1], Constants.Any, doublet =>
258         {
259             var match = Links.SearchOrDefault(sequence[0], doublet);
260             if (match != Constants.Null)
261             {
262                 handler(new LinkAddress<LinkIndex>(match));
263             }
264             return true;
265         });
266         // |_x      ... x_o
267         // |_o      |__|
268         Links.Each(Constants.Any, sequence[0], doublet =>
269         {
270             var match = Links.SearchOrDefault(doublet, sequence[1]);
271             if (match != 0)
272             {
273                 handler(new LinkAddress<LinkIndex>(match));
274             }
275             return true;
276         });
277         //      . _x o _ .
278         //      |__|
279         PartialStepRight(x => handler(x), sequence[0], sequence[1]);
280     }
281     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                     }
438                 filterPosition++;
439             }
440         }
441     });
442 }

```

```

436         return true;
437     });
438     if (filterPosition == sequence.Length)
439     {
440         results.Add(resultIndex);
441     }
442 }
443 if (sequence.Length >= 2)
444 {
445     StepRight(handler, sequence[0], sequence[1]);
446 }
447 var last = sequence.Length - 2;
448 for (var i = 1; i < last; i++)
449 {
450     PartialStepRight(handler, sequence[i], sequence[i + 1]);
451 }
452 if (sequence.Length >= 3)
453 {
454     StepLeft(handler, sequence[sequence.Length - 2],
455         ↪ sequence[sequence.Length - 1]);
456 }
457 }
458 return results;
459 });
460 }
461
462 [MethodImpl(MethodImplOptions.AggressiveInlining)]
463 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
464 {
465     return _sync.ExecuteReadOperation(() =>
466     {
467         var results = new HashSet<ulong>();
468         if (sequence.Length > 0)
469         {
470             Links.EnsureLinkExists(sequence);
471             var firstElement = sequence[0];
472             if (sequence.Length == 1)
473             {
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
507 public const int MaxSequenceFormatSize = 200;
508
509 [MethodImpl(MethodImplOptions.AggressiveInlining)]
510 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
511     ↪ => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);

```

```

510 [MethodImpl(MethodImplOptions.AggressiveInlining)]
511 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
    ↳ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    ↳ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
    ↳ elementToString, insertComma, knownElements));

512
513 [MethodImpl(MethodImplOptions.AggressiveInlining)]
514 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    ↳ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
    ↳ LinkIndex[] knownElements)
515 {
516     var linksInSequence = new HashSet<ulong>(knownElements);
517     //var entered = new HashSet<ulong>();
518     var sb = new StringBuilder();
519     sb.Append('{');
520     if (links.Exists(sequenceLink))
521     {
522         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
523             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
    ↳ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
524         {
525             if (insertComma && sb.Length > 1)
526             {
527                 sb.Append(',');
528             }
529             //if (entered.Contains(element))
530             //{
531             //    sb.Append('{');
532             //    elementToString(sb, element);
533             //    sb.Append('}');
534             //}
535             //else
536             elementToString(sb, element);
537             if (sb.Length < MaxSequenceFormatSize)
538             {
539                 return true;
540             }
541             sb.Append(insertComma ? ", ..." : "...");
542             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,
612                     ↪ Links.Unsync.GetTarget,
613                     x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
614                     ↪ x =>
615                     {
616                         if (filterPosition == (sequence.Length - 1))
617                         {
618                             return false;
619                         }
620                         if (filterPosition >= 0)
621                         {
622                             if (x == sequence[filterPosition + 1])
623                             {
624                                 filterPosition++;
625                             }
626                             else
627                             {
628                                 return false;
629                             }
630                         }
631                         if (filterPosition < 0)
632                         {
633                             if (x == sequence[0])
634                             {
635                                 filterPosition = 0;
636                             }
637                         }
638                         return true;
639                     }
640                 );
641                 if (filterPosition == (sequence.Length - 1))
642                 {
643                     filteredResults.Add(result);
644                 }
645             }
646             return filteredResults;
647         }
648         return new List<ulong>();
649     });
650 }
651
652 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

650 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
651 {
652     return _sync.ExecuteReadOperation(() =>
653     {
654         if (sequence.Length > 0)
655         {
656             Links.EnsureLinkExists(sequence);
657             var results = new HashSet<ulong>();
658             for (var i = 0; i < sequence.Length; i++)
659             {
660                 AllUsagesCore(sequence[i], results);
661             }
662             var filteredResults = new HashSet<ulong>();
663             var matcher = new Matcher(this, sequence, filteredResults, null);
664             matcher.AddAllPartialMatchedToResults(results);
665             return filteredResults;
666         }
667         return new HashSet<ulong>();
668     });
669 }
670
671 [MethodImpl(MethodImplOptions.AggressiveInlining)]
672 public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
673     ↪ params ulong[] sequence)
674 {
675     return _sync.ExecuteReadOperation(() =>
676     {
677         if (sequence.Length > 0)
678         {
679             Links.EnsureLinkExists(sequence);
680
681             var results = new HashSet<ulong>();
682             var filteredResults = new HashSet<ulong>();
683             var matcher = new Matcher(this, sequence, filteredResults, handler);
684             for (var i = 0; i < sequence.Length; i++)
685             {
686                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
687                 {
688                     return false;
689                 }
690             }
691             return true;
692         }
693         return true;
694     });
695 }
696
697 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
698 //{
699 //    return Sync.ExecuteReadOperation(() =>
700 //    {
701 //        if (sequence.Length > 0)
702 //        {
703 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
704 //
705 //            var firstResults = new HashSet<ulong>();
706 //            var lastResults = new HashSet<ulong>();
707 //
708 //            var first = sequence.First(x => x != LinksConstants.Any);
709 //            var last = sequence.Last(x => x != LinksConstants.Any);
710 //
711 //            AllUsagesCore(first, firstResults);
712 //            AllUsagesCore(last, lastResults);
713 //
714 //            firstResults.IntersectWith(lastResults);
715 //
716 //            //for (var i = 0; i < sequence.Length; i++)
717 //            //    AllUsagesCore(sequence[i], results);
718 //
719 //            var filteredResults = new HashSet<ulong>();
720 //            var matcher = new Matcher(this, sequence, filteredResults, null);
721 //            matcher.AddAllPartialMatchedToResults(firstResults);
722 //            return filteredResults;
723 //        }
724 //
725 //        return new HashSet<ulong>();
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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
755 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
756     ↪ IList<ulong> sequence)
757 {
758     return _sync.ExecuteReadOperation(() =>
759     {
760         if (sequence.Count > 0)
761         {
762             Links.EnsureLinkExists(sequence);
763             var results = new HashSet<LinkIndex>();
764             //var nextResults = new HashSet<ulong>();
765             //for (var i = 0; i < sequence.Length; i++)
766             //{
767             //    AllUsagesCore(sequence[i], nextResults);
768             //    if (results.IsNullOrEmpty())
769             //    {
770             //        results = nextResults;
771             //        nextResults = new HashSet<ulong>();
772             //    }
773             //    else
774             //    {
775             //        results.IntersectWith(nextResults);
776             //        nextResults.Clear();
777             //    }
778             //}
779             var collector1 = new AllUsagesCollector1(Links.Unsync, results);
780             collector1.Collect(Links.Unsync.GetLink(sequence[0]));
781             var next = new HashSet<ulong>();
782             for (var i = 1; i < sequence.Count; i++)
783             {
784                 var collector = new AllUsagesCollector1(Links.Unsync, next);
785                 collector.Collect(Links.Unsync.GetLink(sequence[i]));
786
787                 results.IntersectWith(next);
788                 next.Clear();
789             }
790             var filteredResults = new HashSet<ulong>();
791             var matcher = new Matcher(this, sequence, filteredResults, null,
792                 ↪ readAsElements);
793             matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
794                 ↪ x)); // OrderBy is a Hack
795             return filteredResults;
796         }
797         return new HashSet<ulong>();
798     }));
799 }
800 // Does not work
801 //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
802     ↪ params ulong[] sequence)

```



```

800 // {
801 //     var visited = new HashSet<ulong>();
802 //     var results = new HashSet<ulong>();
803 //     var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
    ↪ true; }, readAsElements);
804 //     var last = sequence.Length - 1;
805 //     for (var i = 0; i < last; i++)
806 //     {
807 //         PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
808 //     }
809 //     return results;
810 // }
811
812 [MethodImpl(MethodImplOptions.AggressiveInlining)]
813 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
814 {
815     return _sync.ExecuteReadOperation(() =>
816     {
817         if (sequence.Length > 0)
818         {
819             Links.EnsureLinkExists(sequence);
820             //var firstElement = sequence[0];
821             //if (sequence.Length == 1)
822             //{
823             //    //results.Add(firstElement);
824             //    return results;
825             //}
826             //if (sequence.Length == 2)
827             //{
828             //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
829             //    //if (doublet != Doublets.Links.Null)
830             //    //    results.Add(doublet);
831             //    return results;
832             //}
833             //var lastElement = sequence[sequence.Length - 1];
834             //Func<ulong, bool> handler = x =>
835             //{
836             //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
837             //        ↪ results.Add(x);
838             //    return true;
839             //};
840             //if (sequence.Length >= 2)
841             //    StepRight(handler, sequence[0], sequence[1]);
842             //var last = sequence.Length - 2;
843             //for (var i = 1; i < last; i++)
844             //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
845             //if (sequence.Length >= 3)
846             //    StepLeft(handler, sequence[sequence.Length - 2],
847             //        ↪ sequence[sequence.Length - 1]);
848             //if (sequence.Length == 1)
849             //{
850             //    throw new NotImplementedException(); // all sequences, containing
851             //        ↪ this element?
852             //}
853             //if (sequence.Length == 2)
854             //{
855             //    var results = new List<ulong>();
856             //    PartialStepRight(results.Add, sequence[0], sequence[1]);
857             //    return results;
858             //}
859             //var matches = new List<List<ulong>>();
860             //var last = sequence.Length - 1;
861             //for (var i = 0; i < last; i++)
862             //{
863             //    var results = new List<ulong>();
864             //    //StepRight(results.Add, sequence[i], sequence[i + 1]);
865             //    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
866             //    if (results.Count > 0)
867             //        matches.Add(results);
868             //    else
869             //        return results;
870             //    if (matches.Count == 2)
871             //    {
872                 var merged = new List<ulong>();
873                 for (var j = 0; j < matches[0].Count; j++)
874                     for (var k = 0; k < matches[1].Count; k++)

```

```

872         CloseInnerConnections(merged.Add, matches[0][j],
873         ↪ matches[1][k]);
874         if (merged.Count > 0)
875             matches = new List<List<ulong>> { merged };
876         else
877             return new List<ulong>();
878     }
879     if (matches.Count > 0)
880     {
881         var usages = new HashSet<ulong>();
882         for (int i = 0; i < sequence.Length; i++)
883         {
884             AllUsagesCore(sequence[i], usages);
885         }
886         //for (int i = 0; i < matches[0].Count; i++)
887         //    AllUsagesCore(matches[0][i], usages);
888         //usages.UnionWith(matches[0]);
889         return usages.ToList();
890     }
891     var firstLinkUsages = new HashSet<ulong>();
892     AllUsagesCore(sequence[0], firstLinkUsages);
893     firstLinkUsages.Add(sequence[0]);
894     //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
895     ↪ sequence[0] }; // or all sequences, containing this element?
896     //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
897     ↪ 1).ToList();
898     var results = new HashSet<ulong>();
899     foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
900     ↪ firstLinkUsages, 1))
901     {
902         AllUsagesCore(match, results);
903     }
904     return results.ToList();
905 }
906
907 /// <remarks>
908 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
909 /// </remarks>
910 [MethodImpl(MethodImplOptions.AggressiveInlining)]
911 public HashSet<ulong> AllUsages(ulong link)
912 {
913     return _sync.ExecuteReadOperation(() =>
914     {
915         var usages = new HashSet<ulong>();
916         AllUsagesCore(link, usages);
917         return usages;
918     });
919 }
920
921 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
922 ↪ той связи с которой начинался поиск (STTTSSSTT),
923 // причём достаточно одного бита для хранения перехода влево или вправо
924 [MethodImpl(MethodImplOptions.AggressiveInlining)]
925 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
926 {
927     bool handler(ulong doublet)
928     {
929         if (usages.Add(doublet))
930         {
931             AllUsagesCore(doublet, usages);
932         }
933         return true;
934     }
935     Links.Unsync.Each(link, Constants.Any, handler);
936     Links.Unsync.Each(Constants.Any, link, handler);
937 }
938
939 [MethodImpl(MethodImplOptions.AggressiveInlining)]
940 public HashSet<ulong> AllBottomUsages(ulong link)
941 {
942     return _sync.ExecuteReadOperation(() =>
943     {
944         var visits = new HashSet<ulong>();

```

```

944     var usages = new HashSet<ulong>();
945     AllBottomUsagesCore(link, visits, usages);
946     return usages;
947 });
948 }
949
950 [MethodImpl(MethodImplOptions.AggressiveInlining)]
951 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
    ↳ usages)
952 {
953     bool handler(ulong doublet)
954     {
955         if (visits.Add(doublet))
956         {
957             AllBottomUsagesCore(doublet, visits, usages);
958         }
959         return true;
960     }
961     if (Links.Unsync.Count(Constants.Any, link) == 0)
962     {
963         usages.Add(link);
964     }
965     else
966     {
967         Links.Unsync.Each(link, Constants.Any, handler);
968         Links.Unsync.Each(Constants.Any, link, handler);
969     }
970 }
971
972 [MethodImpl(MethodImplOptions.AggressiveInlining)]
973 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
974 {
975     if (Options.UseSequenceMarker)
976     {
977         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
    ↳ Options.MarkedSequenceMatcher, symbol);
978         return counter.Count();
979     }
980     else
981     {
982         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
    ↳ symbol);
983         return counter.Count();
984     }
985 }
986
987 [MethodImpl(MethodImplOptions.AggressiveInlining)]
988 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
    ↳ LinkIndex> outerHandler)
989 {
990     bool handler(ulong doublet)
991     {
992         if (usages.Add(doublet))
993         {
994             if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
995             {
996                 return false;
997             }
998             if (!AllUsagesCore1(doublet, usages, outerHandler))
999             {
1000                 return false;
1001             }
1002         }
1003         return true;
1004     }
1005     return Links.Unsync.Each(link, Constants.Any, handler)
    && Links.Unsync.Each(Constants.Any, link, handler);
1006 }
1007
1008
1009 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1010 public void CalculateAllUsages(ulong[] totals)
1011 {
1012     var calculator = new AllUsagesCalculator(Links, totals);
1013     calculator.Calculate();
1014 }
1015
1016 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1017 public void CalculateAllUsages2(ulong[] totals)

```

```

1018 {
1019     var calculator = new AllUsagesCalculator2(Links, totals);
1020     calculator.Calculate();
1021 }
1022
1023 private class AllUsagesCalculator
1024 {
1025     private readonly SynchronizedLinks<ulong> _links;
1026     private readonly ulong[] _totals;
1027
1028     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1029     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
1030     {
1031         _links = links;
1032         _totals = totals;
1033     }
1034
1035     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1036     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
        ↪ CalculateCore);
1037
1038     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1039     private bool CalculateCore(ulong link)
1040     {
1041         if (_totals[link] == 0)
1042         {
1043             var total = 1UL;
1044             _totals[link] = total;
1045             var visitedChildren = new HashSet<ulong>();
1046             bool linkCalculator(ulong child)
1047             {
1048                 if (link != child && visitedChildren.Add(child))
1049                 {
1050                     total += _totals[child] == 0 ? 1 : _totals[child];
1051                 }
1052                 return true;
1053             }
1054             _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1055             _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1056             _totals[link] = total;
1057         }
1058         return true;
1059     }
1060 }
1061
1062 private class AllUsagesCalculator2
1063 {
1064     private readonly SynchronizedLinks<ulong> _links;
1065     private readonly ulong[] _totals;
1066
1067     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1068     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1069     {
1070         _links = links;
1071         _totals = totals;
1072     }
1073
1074     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1075     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
        ↪ CalculateCore);
1076
1077     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1078     private bool IsElement(ulong link)
1079     {
1080         // _linksInSequence.Contains(link) ||
1081         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
        ↪ link;
1082     }
1083
1084     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1085     private bool CalculateCore(ulong link)
1086     {
1087         // TODO: Проработать защиту от заикливания
1088         // Основано на SequenceWalker.WalkLeft
1089         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1090         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1091         Func<ulong, bool> isElement = IsElement;
1092         void visitLeaf(ulong parent)
1093     {

```

```

1094         if (link != parent)
1095         {
1096             _totals[parent]++;
1097         }
1098     }
1099     void visitNode(ulong parent)
1100     {
1101         if (link != parent)
1102         {
1103             _totals[parent]++;
1104         }
1105     }
1106     var stack = new Stack();
1107     var element = link;
1108     if (isElement(element))
1109     {
1110         visitLeaf(element);
1111     }
1112     else
1113     {
1114         while (true)
1115         {
1116             if (isElement(element))
1117             {
1118                 if (stack.Count == 0)
1119                 {
1120                     break;
1121                 }
1122                 element = stack.Pop();
1123                 var source = getSource(element);
1124                 var target = getTarget(element);
1125                 // Обработка элемента
1126                 if (isElement(target))
1127                 {
1128                     visitLeaf(target);
1129                 }
1130                 if (isElement(source))
1131                 {
1132                     visitLeaf(source);
1133                 }
1134                 element = source;
1135             }
1136             else
1137             {
1138                 stack.Push(element);
1139                 visitNode(element);
1140                 element = getTarget(element);
1141             }
1142         }
1143     }
1144     _totals[link]++;
1145     return true;
1146 }
1147
1148
1149 private class AllUsagesCollector
1150 {
1151     private readonly ILinks<ulong> _links;
1152     private readonly HashSet<ulong> _usages;
1153
1154     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1155     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1156     {
1157         _links = links;
1158         _usages = usages;
1159     }
1160
1161     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1162     public bool Collect(ulong link)
1163     {
1164         if (_usages.Add(link))
1165         {
1166             _links.Each(link, _links.Constants.Any, Collect);
1167             _links.Each(_links.Constants.Any, link, Collect);
1168         }
1169         return true;
1170     }
1171 }
1172

```

```

1173 private class AllUsagesCollector1
1174 {
1175     private readonly ILinks<ulong> _links;
1176     private readonly HashSet<ulong> _usages;
1177     private readonly ulong _continue;
1178
1179     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1180     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1181     {
1182         _links = links;
1183         _usages = usages;
1184         _continue = _links.Constants.Continue;
1185     }
1186
1187     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1188     public ulong Collect(IList<ulong> link)
1189     {
1190         var linkIndex = _links.GetIndex(link);
1191         if (_usages.Add(linkIndex))
1192         {
1193             _links.Each(Collect, _links.Constants.Any, linkIndex);
1194         }
1195         return _continue;
1196     }
1197 }
1198
1199 private class AllUsagesCollector2
1200 {
1201     private readonly ILinks<ulong> _links;
1202     private readonly BitString _usages;
1203
1204     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1205     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1206     {
1207         _links = links;
1208         _usages = usages;
1209     }
1210
1211     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1212     public bool Collect(ulong link)
1213     {
1214         if (_usages.Add((long)link))
1215         {
1216             _links.Each(link, _links.Constants.Any, Collect);
1217             _links.Each(_links.Constants.Any, link, Collect);
1218         }
1219         return true;
1220     }
1221 }
1222
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,
    ↪ secondLinkUsage);
1302             StepRight(filler.AddFirstAndReturnConstant, previousMatching,
    ↪ secondLinkUsage);
1303             TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
    ↪ previousMatching);
1304             //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
    ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
    ↪ желаемым результатам.
1305             PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
    ↪ secondLinkUsage);
1306         }
1307     }
1308     if (matchings.Count == 0)
1309     {
1310         return matchings;
1311     }
1312     return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1313 }
1314
1315 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1316 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
    ↪ links, params ulong[] sequence)
1317 {
1318     if (sequence == null)

```

```

1319     {
1320         return;
1321     }
1322     for (var i = 0; i < sequence.Length; i++)
1323     {
1324         if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
1325             ↪ !links.Exists(sequence[i]))
1326         {
1327             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1328                 ↪ $"patternSequence[{i}]");
1329         }
1330     }
1331 }
1332
1333 // Pattern Matching -> Key To Triggers
1334 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1335 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1336 {
1337     return _sync.ExecuteReadOperation(() =>
1338     {
1339         patternSequence = Simplify(patternSequence);
1340         if (patternSequence.Length > 0)
1341         {
1342             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1343             var uniqueSequenceElements = new HashSet<ulong>();
1344             for (var i = 0; i < patternSequence.Length; i++)
1345             {
1346                 if (patternSequence[i] != Constants.Any && patternSequence[i] !=
1347                     ↪ ZeroOrMany)
1348                 {
1349                     uniqueSequenceElements.Add(patternSequence[i]);
1350                 }
1351             }
1352             var results = new HashSet<ulong>();
1353             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1354             {
1355                 AllUsagesCore(uniqueSequenceElement, results);
1356             }
1357             var filteredResults = new HashSet<ulong>();
1358             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1359             matcher.AddAllPatternMatchedToResults(results);
1360             return filteredResults;
1361         }
1362         return new HashSet<ulong>();
1363     });
1364 }
1365
1366 // Найти все возможные связи между указанным списком связей.
1367 // Находит связи между всеми указанными связями в любом порядке.
1368 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
1369 ↪ несколько раз в последовательности)
1370 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1371 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1372 {
1373     return _sync.ExecuteReadOperation(() =>
1374     {
1375         var results = new HashSet<ulong>();
1376         if (linksToConnect.Length > 0)
1377         {
1378             Links.EnsureLinkExists(linksToConnect);
1379             AllUsagesCore(linksToConnect[0], results);
1380             for (var i = 1; i < linksToConnect.Length; i++)
1381             {
1382                 var next = new HashSet<ulong>();
1383                 AllUsagesCore(linksToConnect[i], next);
1384                 results.IntersectWith(next);
1385             }
1386             return results;
1387         }
1388     });
1389 }
1390
1391 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1392 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1393 {
1394     return _sync.ExecuteReadOperation(() =>
1395     {
1396         var results = new HashSet<ulong>();

```



```

1393     if (linksToConnect.Length > 0)
1394     {
1395         Links.EnsureLinkExists(linksToConnect);
1396         var collector1 = new AllUsagesCollector(Links.Unsync, results);
1397         collector1.Collect(linksToConnect[0]);
1398         var next = new HashSet<ulong>();
1399         for (var i = 1; i < linksToConnect.Length; i++)
1400         {
1401             var collector = new AllUsagesCollector(Links.Unsync, next);
1402             collector.Collect(linksToConnect[i]);
1403             results.IntersectWith(next);
1404             next.Clear();
1405         }
1406     }
1407     return results;
1408 });
1409 }
1410
1411 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1412 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1413 {
1414     return _sync.ExecuteReadOperation(() =>
1415     {
1416         var results = new HashSet<ulong>();
1417         if (linksToConnect.Length > 0)
1418         {
1419             Links.EnsureLinkExists(linksToConnect);
1420             var collector1 = new AllUsagesCollector(Links, results);
1421             collector1.Collect(linksToConnect[0]);
1422             //AllUsagesCore(linksToConnect[0], results);
1423             for (var i = 1; i < linksToConnect.Length; i++)
1424             {
1425                 var next = new HashSet<ulong>();
1426                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1427                 collector.Collect(linksToConnect[i]);
1428                 //AllUsagesCore(linksToConnect[i], next);
1429                 //results.IntersectWith(next);
1430                 results = next;
1431             }
1432         }
1433         return results;
1434     });
1435 }
1436
1437 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1438 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1439 {
1440     return _sync.ExecuteReadOperation(() =>
1441     {
1442         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1443         ↪ BitArray((int)_links.Total + 1);
1444         if (linksToConnect.Length > 0)
1445         {
1446             Links.EnsureLinkExists(linksToConnect);
1447             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1448             collector1.Collect(linksToConnect[0]);
1449             for (var i = 1; i < linksToConnect.Length; i++)
1450             {
1451                 var next = new BitString((long)Links.Unsync.Count() + 1); //new
1452                 ↪ BitArray((int)_links.Total + 1);
1453                 var collector = new AllUsagesCollector2(Links.Unsync, next);
1454                 collector.Collect(linksToConnect[i]);
1455                 results = results.And(next);
1456             }
1457             return results.GetSetUInt64Indices();
1458         }
1459     });
1460 }
1461
1462 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1463 private static ulong[] Simplify(ulong[] sequence)
1464 {
1465     // Считаем новый размер последовательности
1466     long newLength = 0;
1467     var zeroOrManyStepped = false;
1468     for (var i = 0; i < sequence.Length; i++)
1469     {
1470         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;

```

```

1546     var rightBound = links.Length - 1;
1547     var left = links[leftBound++];
1548     var right = links[rightBound--];
1549     var results = new List<ulong>();
1550     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1551     return results;
1552 }
1553
1554 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1555 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
    ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1556 {
1557     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1558     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1559     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1560     {
1561         var nextLeftLink = middleLinks[leftBound];
1562         var elements = GetRightElements(leftLink, nextLeftLink);
1563         if (leftBound <= rightBound)
1564         {
1565             for (var i = elements.Length - 1; i >= 0; i--)
1566             {
1567                 var element = elements[i];
1568                 if (element != 0)
1569                 {
1570                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
1571                         ↪ rightLink, rightBound, ref results);
1572                 }
1573             }
1574         }
1575     }
1576     else
1577     {
1578         for (var i = elements.Length - 1; i >= 0; i--)
1579         {
1580             var element = elements[i];
1581             if (element != 0)
1582             {
1583                 results.Add(element);
1584             }
1585         }
1586     }
1587 }
1588 else
1589 {
1590     var nextRightLink = middleLinks[rightBound];
1591     var elements = GetLeftElements(rightLink, nextRightLink);
1592     if (leftBound <= rightBound)
1593     {
1594         for (var i = elements.Length - 1; i >= 0; i--)
1595         {
1596             var element = elements[i];
1597             if (element != 0)
1598             {
1599                 CollectMatchingSequences(leftLink, leftBound, middleLinks,
1600                     ↪ elements[i], rightBound - 1, ref results);
1601             }
1602         }
1603     }
1604     else
1605     {
1606         for (var i = elements.Length - 1; i >= 0; i--)
1607         {
1608             var element = elements[i];
1609             if (element != 0)
1610             {
1611                 results.Add(element);
1612             }
1613         }
1614     }
1615 }
1616 }
1617
1618 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1619 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1620 {
1621     var result = new ulong[5];
1622     TryStepRight(startLink, rightLink, result, 0);
1623     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             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             ↪ == Net.And &&
1711         {
1712             result[offset + 1] = couple;
1713             if (++added == 2)
1714             {
1715                 return false;
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
1743     struct PatternBlock
1744     {
1745         public PatternBlockType Type;
1746         public long Start;
1747         public long Stop;
1748     }
1749
1750     private readonly List<PatternBlock> _pattern;
1751     private int _patternPosition;
1752     private long _sequencePosition;
1753
1754     #endregion
1755
1756     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1757     public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1758         ↪ HashSet<LinkIndex> results)
1759         : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1760     {
1761         _sequences = sequences;
1762         _patternSequence = patternSequence;
1763         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1764             ↪ _sequences.Constants.Any && x != ZeroOrMany));
1765         _results = results;
1766         _pattern = CreateDetailedPattern();
1767     }
1768
1769     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1770     protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
1771         ↪ base.IsElement(link);
1772
1773     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1774     public bool PatternMatch(LinkIndex sequenceToMatch)
1775     {
1776         _patternPosition = 0;
1777         _sequencePosition = 0;
1778         foreach (var part in Walk(sequenceToMatch))

```

```

1776         {
1777             if (!PatternMatchCore(part))
1778             {
1779                 break;
1780             }
1781         }
1782         return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1783             ↪ - 1 && _pattern[_patternPosition].Start == 0);
1784     }
1785     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1786     private List<PatternBlock> CreateDetailedPattern()
1787     {
1788         var pattern = new List<PatternBlock>();
1789         var patternBlock = new PatternBlock();
1790         for (var i = 0; i < _patternSequence.Length; i++)
1791         {
1792             if (patternBlock.Type == PatternBlockType.Undefined)
1793             {
1794                 if (_patternSequence[i] == _sequences.Constants.Any)
1795                 {
1796                     patternBlock.Type = PatternBlockType.Gap;
1797                     patternBlock.Start = 1;
1798                     patternBlock.Stop = 1;
1799                 }
1800                 else if (_patternSequence[i] == ZeroOrMany)
1801                 {
1802                     patternBlock.Type = PatternBlockType.Gap;
1803                     patternBlock.Start = 0;
1804                     patternBlock.Stop = long.MaxValue;
1805                 }
1806                 else
1807                 {
1808                     patternBlock.Type = PatternBlockType.Elements;
1809                     patternBlock.Start = i;
1810                     patternBlock.Stop = i;
1811                 }
1812             }
1813             else if (patternBlock.Type == PatternBlockType.Elements)
1814             {
1815                 if (_patternSequence[i] == _sequences.Constants.Any)
1816                 {
1817                     pattern.Add(patternBlock);
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 //    {
1901 //        /* a * matches zero or more instances */
1902 //        if (matchhere(regexp, text))
1903 //            return 1;
1904 //    } while (*text != '\0' && (*text++ == c || c == '.'));
1905 //    return 0;
1906 //}
1907
1908 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
1909 ↪ long maximumGap)
1910 //{
1911 //    mininumGap = 0;
1912 //    maximumGap = 0;
1913 //    element = 0;
1914 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1915 //    {
1916 //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1917 //            mininumGap++;
1918 //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1919 //            maximumGap = long.MaxValue;
1920 //        else
1921 //            break;
1922 //    }
1923
1924 //    if (maximumGap < mininumGap)
1925 //        maximumGap = mininumGap;
1926 //}
1927
1928 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1929 private bool PatternMatchCore(LinkIndex element)
1930 {
1931     if (_patternPosition >= _pattern.Count)
1932     {
1933         _patternPosition = -2;
1934         return false;
1935     }

```

```

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 {
1975     var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1976     if (_patternSequence[patternElementPosition] != element)
1977     {
1978         return false; // Соответствие невозможно
1979     }
1980     if (patternElementPosition == currentPatternBlock.Stop)
1981     {
1982         _patternPosition++;
1983         _sequencePosition = 0;
1984     }
1985     else
1986     {
1987         _sequencePosition++;
1988     }
1989 }
1990 return true;
1991 //if (_patternSequence[_patternPosition] != element)
1992 //    return false;
1993 //else
1994 //{
1995 //    _sequencePosition++;
1996 //    _patternPosition++;
1997 //    return true;
1998 //}
1999 //if (_filterPosition == _patternSequence.Length)
2000 //{
2001 //    _filterPosition = -2; // Длиннее чем нужно
2002 //    return false;
2003 //}
2004 //if (element != _patternSequence[_filterPosition])
2005 //{
2006 //    _filterPosition = -1;
2007 //    return false; // Начинается иначе
2008 //}
2009 //_filterPosition++;
2010 //if (_filterPosition == (_patternSequence.Length - 1))

```



```

2011         //     return false;
2012         //if (_filterPosition >= 0)
2013         //{
2014         //     if (element == _patternSequence[_filterPosition + 1])
2015         //         _filterPosition++;
2016         //     else
2017         //         return false;
2018         //}
2019         //if (_filterPosition < 0)
2020         //{
2021         //     if (element == _patternSequence[0])
2022         //         _filterPosition = 0;
2023         //}
2024     }
2025
2026     [MethodImpl(MethodImplOptions.AggressiveInlining)]
2027     public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
2028     {
2029         foreach (var sequenceToMatch in sequencesToMatch)
2030         {
2031             if (PatternMatch(sequenceToMatch))
2032             {
2033                 _results.Add(sequenceToMatch);
2034             }
2035         }
2036     }
2037 }
2038
2039 #endregion
2040 }
2041 }

```

#### 1.117 ./csharp/Platform.Data.Doublets/Sequences/Sequences.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections;
6  using Platform.Collections.Lists;
7  using Platform.Collections.Stacks;
8  using Platform.Threading.Synchronization;
9  using Platform.Data.Doublets.Sequences.Walkers;
10 using LinkIndex = System.UInt64;
11
12 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     /// <summary>
17     /// Представляет коллекцию последовательностей связей.
18     /// </summary>
19     /// <remarks>
20     /// Обязательно реализовать атомарность каждого публичного метода.
21     ///
22     /// TODO:
23     ///
24     /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
25     /// через естественную группировку по unicode типам, все whitespace вместе, все символы
26     ///     ↳ вместе, все числа вместе и т.п.
27     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
28     ///     ↳ графа)
29     ///
30     /// х*у - найти все связи между, в последовательностях любой формы, если не стоит
31     ///     ↳ ограничитель на то, что является последовательностью, а что нет,
32     /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
33     ///     ↳ порядке.
34     ///
35     /// Рост последовательности слева и справа.
36     /// Поиск со звездочкой.
37     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
38     /// так же проблема может быть решена при реализации дистанционных триггеров.
39     /// Нужны ли уникальные указатели вообще?
40     /// Что если обращение к информации будет происходить через содержимое всегда?
41     ///
42     /// Писать тесты.
43     ///
44     ///
45     /// Можно убрать зависимость от конкретной реализации Links,
46     /// на зависимость от абстрактного элемента, который может быть представлен несколькими
47     ///     ↳ способами.

```

```

43 ///
44 /// Можно ли как-то сделать один общий интерфейс
45 ///
46 ///
47 /// Блокчейн и/или гит для распределённой записи транзакций.
48 ///
49 /// </remarks>
50 public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
    ↪ (после завершения реализации Sequences)
51 {
52     /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
    ↪ связей.</summary>
53     public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
54
55     public SequencesOptions<LinkIndex> Options { get; }
56     public SynchronizedLinks<LinkIndex> Links { get; }
57     private readonly ISynchronization _sync;
58
59     public LinksConstants<LinkIndex> Constants { get; }
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
63     {
64         Links = links;
65         _sync = links.SyncRoot;
66         Options = options;
67         Options.ValidateOptions();
68         Options.InitOptions(Links);
69         Constants = links.Constants;
70     }
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
    ↪ SequencesOptions<LinkIndex>()) { }
74
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     public bool IsSequence(LinkIndex sequence)
77     {
78         return _sync.ExecuteReadOperation(() =>
79         {
80             if (Options.UseSequenceMarker)
81             {
82                 return Options.MarkedSequenceMatcher.IsMatched(sequence);
83             }
84             return !Links.Unsync.IsPartialPoint(sequence);
85         });
86     }
87
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     private LinkIndex GetSequenceByElements(LinkIndex sequence)
90     {
91         if (Options.UseSequenceMarker)
92         {
93             return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
94         }
95         return sequence;
96     }
97
98     [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
176 #region Create
177
178 [MethodImpl(MethodImplOptions.AggressiveInlining)]
179 public LinkIndex Create(ICollection<LinkIndex> restrictions)
180 {
181     return _sync.ExecuteWriteOperation(() =>
182     {
183         if (restrictions.IsNullOrEmpty())
184         {
185             return Constants.Null;
186         }
187         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
188         return CreateCore(restrictions);
189     });
190 }
191
192 [MethodImpl(MethodImplOptions.AggressiveInlining)]
193 private LinkIndex CreateCore(ICollection<LinkIndex> restrictions)
194 {
195     LinkIndex[] sequence = restrictions.SkipFirst();
196     if (Options.UseIndex)

```

```

197     {
198         Options.Index.Add(sequence);
199     }
200     var sequenceRoot = default(LinkIndex);
201     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
202     {
203         var matches = Each(restrictions);
204         if (matches.Count > 0)
205         {
206             sequenceRoot = matches[0];
207         }
208     }
209     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
210     {
211         return CompactCore(sequence);
212     }
213     if (sequenceRoot == default)
214     {
215         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
216     }
217     if (Options.UseSequenceMarker)
218     {
219         return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
220     }
221     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
222 }
223
224 #endregion
225
226 #region Each
227
228 [MethodImpl(MethodImplOptions.AggressiveInlining)]
229 public List<LinkIndex> Each(IList<LinkIndex> sequence)
230 {
231     var results = new List<LinkIndex>();
232     var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
233     Each(filler.AddFirstAndReturnConstant, sequence);
234     return results;
235 }
236
237 [MethodImpl(MethodImplOptions.AggressiveInlining)]
238 public LinkIndex Each(Func<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,
256                         ↪ Options.SequenceMarkerLink, any));
257                 }
258                 else
259                 {
260                     return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
261                         ↪ any));
262                 }
263             }
264             if (Options.UseSequenceMarker)
265             {
266                 var sequenceLinkValues = Links.Unsync.GetLink(link);
267                 if (sequenceLinkValues[Constants.SourcePart] ==
268                     ↪ Options.SequenceMarkerLink)
269                 {
270                     link = sequenceLinkValues[Constants.TargetPart];
271                 }
272             }
273             var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();

```

```

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]) !=
    ↪ Constants.Continue)
309         {
310             return Constants.Break;
311         }
312     }
313     if (values.Count >= 3)
314     {
315         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
    ↪ != Constants.Continue)
316         {
317             return Constants.Break;
318         }
319     }
320     return Constants.Continue;
321 }
322
323 [MethodImpl(MethodImplOptions.AggressiveInlining)]
324 private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↪ left, LinkIndex right)
325 {
326     return Links.Unsync.Each(doublet =>
327     {
328         var doubletIndex = doublet[Constants.IndexPart];
329         if (StepRight(handler, doubletIndex, right) != Constants.Continue)
330         {
331             return Constants.Break;
332         }
333         if (left != doubletIndex)
334         {
335             return PartialStepRight(handler, doubletIndex, right);
336         }
337         return Constants.Continue;
338     }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
339 }
340
341 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

342 private LinkIndex StepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↳ LinkIndex right) => Links.Unsync.Each(rightStep => TryStepRightUp(handler, right,
    ↳ rightStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, left,
    ↳ Constants.Any));
343
344 [MethodImpl(MethodImplOptions.AggressiveInlining)]
345 private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ right, LinkIndex stepFrom)
346 {
347     var upStep = stepFrom;
348     var firstSource = Links.Unsync.GetTarget(upStep);
349     while (firstSource != right && firstSource != upStep)
350     {
351         upStep = firstSource;
352         firstSource = Links.Unsync.GetSource(upStep);
353     }
354     if (firstSource == right)
355     {
356         return handler(new LinkAddress<LinkIndex>(stepFrom));
357     }
358     return Constants.Continue;
359 }
360
361 [MethodImpl(MethodImplOptions.AggressiveInlining)]
362 private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↳ LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
    ↳ leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
    ↳ right));
363
364 [MethodImpl(MethodImplOptions.AggressiveInlining)]
365 private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ left, LinkIndex stepFrom)
366 {
367     var upStep = stepFrom;
368     var firstTarget = Links.Unsync.GetSource(upStep);
369     while (firstTarget != left && firstTarget != upStep)
370     {
371         upStep = firstTarget;
372         firstTarget = Links.Unsync.GetTarget(upStep);
373     }
374     if (firstTarget == left)
375     {
376         return handler(new LinkAddress<LinkIndex>(stepFrom));
377     }
378     return Constants.Continue;
379 }
380
381 #endregion
382
383 #region Update
384
385 [MethodImpl(MethodImplOptions.AggressiveInlining)]
386 public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
387 {
388     var sequence = restrictions.SkipFirst();
389     var newSequence = substitution.SkipFirst();
390     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
391     {
392         return Constants.Null;
393     }
394     if (sequence.IsNullOrEmpty())
395     {
396         return Create(substitution);
397     }
398     if (newSequence.IsNullOrEmpty())
399     {
400         Delete(restrictions);
401         return Constants.Null;
402     }
403     return _sync.ExecuteWriteOperation((Func<ulong>)((() =>
404     {
405         ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
406         Links.EnsureLinkExists(newSequence);
407         return UpdateCore(sequence, newSequence);
408     })));
409 }
410
411 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

412 private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence)
413 {
414     LinkIndex bestVariant;
415     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
416         ↪ !sequence.EqualTo(newSequence))
417     {
418         bestVariant = CompactCore(newSequence);
419     }
420     else
421     {
422         bestVariant = CreateCore(newSequence);
423     }
424     // TODO: Check all options only ones before loop execution
425     // Возможно нужно две версии Each, возвращающий фактические последовательности и с
426     ↪ маркером,
427     // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
428     ↪ можно получить имея только фактические последовательности.
429     foreach (var variant in Each(sequence))
430     {
431         if (variant != bestVariant)
432         {
433             UpdateOneCore(variant, bestVariant);
434         }
435     }
436     return bestVariant;
437 }
438
439 [MethodImpl(MethodImplOptions.AggressiveInlining)]
440 private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
441 {
442     if (Options.UseGarbageCollection)
443     {
444         var sequenceElements = GetSequenceElements(sequence);
445         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
446         var sequenceLink = GetSequenceByElements(sequenceElements);
447         var newSequenceElements = GetSequenceElements(newSequence);
448         var newSequenceLink = GetSequenceByElements(newSequenceElements);
449         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
450         {
451             if (sequenceLink != Constants.Null)
452             {
453                 Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
454             }
455             Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
456         }
457         ClearGarbage(sequenceElementsContents.Source);
458         ClearGarbage(sequenceElementsContents.Target);
459     }
460     else
461     {
462         if (Options.UseSequenceMarker)
463         {
464             var sequenceElements = GetSequenceElements(sequence);
465             var sequenceLink = GetSequenceByElements(sequenceElements);
466             var newSequenceElements = GetSequenceElements(newSequence);
467             var newSequenceLink = GetSequenceByElements(newSequenceElements);
468             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
469             {
470                 if (sequenceLink != Constants.Null)
471                 {
472                     Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
473                 }
474                 Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
475             }
476         }
477         else
478         {
479             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
480             {
481                 Links.Unsync.MergeAndDelete(sequence, newSequence);
482             }
483         }
484     }
485 }
486
487 #endregion
488
489 #region Delete

```

```

487 [MethodImpl(MethodImplOptions.AggressiveInlining)]
488 public void Delete(ICollection<LinkIndex> restrictions)
489 {
490     _sync.ExecuteWriteOperation(() =>
491     {
492         var sequence = restrictions.SkipFirst();
493         // TODO: Check all options only ones before loop execution
494         foreach (var linkToDelete in Each(sequence))
495         {
496             DeleteOneCore(linkToDelete);
497         }
498     });
499 }
500
501 [MethodImpl(MethodImplOptions.AggressiveInlining)]
502 private void DeleteOneCore(LinkIndex link)
503 {
504     if (Options.UseGarbageCollection)
505     {
506         var sequenceElements = GetSequenceElements(link);
507         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
508         var sequenceLink = GetSequenceByElements(sequenceElements);
509         if (Options.UseCascadeDelete || CountUsages(link) == 0)
510         {
511             if (sequenceLink != Constants.Null)
512             {
513                 Links.Unsync.Delete(sequenceLink);
514             }
515             Links.Unsync.Delete(link);
516         }
517         ClearGarbage(sequenceElementsContents.Source);
518         ClearGarbage(sequenceElementsContents.Target);
519     }
520     else
521     {
522         if (Options.UseSequenceMarker)
523         {
524             var sequenceElements = GetSequenceElements(link);
525             var sequenceLink = GetSequenceByElements(sequenceElements);
526             if (Options.UseCascadeDelete || CountUsages(link) == 0)
527             {
528                 if (sequenceLink != Constants.Null)
529                 {
530                     Links.Unsync.Delete(sequenceLink);
531                 }
532                 Links.Unsync.Delete(link);
533             }
534         }
535         else
536         {
537             if (Options.UseCascadeDelete || CountUsages(link) == 0)
538             {
539                 Links.Unsync.Delete(link);
540             }
541         }
542     }
543 }
544
545 #endregion
546
547 #region Compactification
548
549 [MethodImpl(MethodImplOptions.AggressiveInlining)]
550 public void CompactAll()
551 {
552     _sync.ExecuteWriteOperation(() =>
553     {
554         var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
555         for (int i = 0; i < sequences.Count; i++)
556         {
557             var sequence = this.ToList(sequences[i]);
558             Compact(sequence.ShiftRight());
559         }
560     });
561 }
562
563 /// <remarks>
564

```



```

565 /// bestVariant можно выбирать по максимальному числу использований,
566 /// но сбалансированный 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(IList<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(IList<LinkIndex> sequence) => UpdateCore(sequence,
587     ↪ sequence);
588
589 #endregion
590
591 #region Garbage Collection
592
593 /// <remarks>
594 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
595 ↪ определить извне или в унаследованном классе
596 /// </remarks>
597 [MethodImpl(MethodImplOptions.AggressiveInlining)]
598 private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
599     ↪ !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
600
601 [MethodImpl(MethodImplOptions.AggressiveInlining)]
602 private void ClearGarbage(LinkIndex link)
603 {
604     if (IsGarbage(link))
605     {
606         var contents = new Link<ulong>(Links.GetLink(link));
607         Links.Unsync.Delete(link);
608         ClearGarbage(contents.Source);
609         ClearGarbage(contents.Target);
610     }
611 }
612
613 #endregion
614
615 #region Walkers
616
617 [MethodImpl(MethodImplOptions.AggressiveInlining)]
618 public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
619 {
620     return _sync.ExecuteReadOperation(() =>
621     {
622         var links = Links.Unsync;
623         foreach (var part in Options.Walker.Walk(sequence))
624         {
625             if (!handler(part))
626             {
627                 return false;
628             }
629         }
630         return true;
631     });
632 }
633
634 public class Matcher : RightSequenceWalker<LinkIndex>
635 {
636     private readonly Sequences _sequences;
637     private readonly IList<LinkIndex> _patternSequence;
638     private readonly HashSet<LinkIndex> _linksInSequence;
639     private readonly HashSet<LinkIndex> _results;
640     private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
641     private readonly HashSet<LinkIndex> _readAsElements;
642     private int _filterPosition;
643
644     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

642 public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
↪ HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
↪ HashSet<LinkIndex> readAsElements = null)
643 : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
644 {
645     _sequences = sequences;
646     _patternSequence = patternSequence;
647     _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
↪ _links.Constants.Any && x != ZeroOrMany));
648     _results = results;
649     _stopableHandler = stopableHandler;
650     _readAsElements = readAsElements;
651 }
652
653 [MethodImpl(MethodImplOptions.AggressiveInlining)]
654 protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
↪ (_readAsElements != null && _readAsElements.Contains(link)) ||
↪ _linksInSequence.Contains(link);
655
656 [MethodImpl(MethodImplOptions.AggressiveInlining)]
657 public bool FullMatch(LinkIndex sequenceToMatch)
658 {
659     _filterPosition = 0;
660     foreach (var part in Walk(sequenceToMatch))
661     {
662         if (!FullMatchCore(part))
663         {
664             break;
665         }
666     }
667     return _filterPosition == _patternSequence.Count;
668 }
669
670 [MethodImpl(MethodImplOptions.AggressiveInlining)]
671 private bool FullMatchCore(LinkIndex element)
672 {
673     if (_filterPosition == _patternSequence.Count)
674     {
675         _filterPosition = -2; // Длиннее чем нужно
676         return false;
677     }
678     if (_patternSequence[_filterPosition] != _links.Constants.Any
↪ && element != _patternSequence[_filterPosition])
679     {
680         _filterPosition = -1;
681         return false; // Начинается/Продолжается иначе
682     }
683     _filterPosition++;
684     return true;
685 }
686
687 [MethodImpl(MethodImplOptions.AggressiveInlining)]
688 public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
689 {
690     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
691     if (FullMatch(sequenceToMatch))
692     {
693         _results.Add(sequenceToMatch);
694     }
695 }
696
697 [MethodImpl(MethodImplOptions.AggressiveInlining)]
698 public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
699 {
700     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
701     if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
702     {
703         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
704     }
705     return _links.Constants.Continue;
706 }
707
708 [MethodImpl(MethodImplOptions.AggressiveInlining)]
709 public LinkIndex HandleFullMatchedSequence(IList<LinkIndex> restrictions)
710 {
711     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
712     var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
713     if (sequence != _links.Constants.Null && FullMatch(sequenceToMatch) &&
↪ _results.Add(sequenceToMatch))
714

```

```

715     {
716         return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
717     }
718     return _links.Constants.Continue;
719 }
720
721 /// <remarks>
722 /// TODO: Add support for LinksConstants.Any
723 /// </remarks>
724 [MethodImpl(MethodImplOptions.AggressiveInlining)]
725 public bool PartialMatch(LinkIndex sequenceToMatch)
726 {
727     _filterPosition = -1;
728     foreach (var part in Walk(sequenceToMatch))
729     {
730         if (!PartialMatchCore(part))
731         {
732             break;
733         }
734     }
735     return _filterPosition == _patternSequence.Count - 1;
736 }
737
738 [MethodImpl(MethodImplOptions.AggressiveInlining)]
739 private bool PartialMatchCore(LinkIndex element)
740 {
741     if (_filterPosition == (_patternSequence.Count - 1))
742     {
743         return false; // Нашлось
744     }
745     if (_filterPosition >= 0)
746     {
747         if (element == _patternSequence[_filterPosition + 1])
748         {
749             _filterPosition++;
750         }
751         else
752         {
753             _filterPosition = -1;
754         }
755     }
756     if (_filterPosition < 0)
757     {
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 }

```

```

794     }
795 }
796 }
797
798 [MethodImpl(MethodImplOptions.AggressiveInlining)]
799 public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
    ↪ sequencesToMatch)
800 {
801     foreach (var sequenceToMatch in sequencesToMatch)
802     {
803         if (PartialMatch(sequenceToMatch))
804         {
805             _readAsElements.Add(sequenceToMatch);
806             _results.Add(sequenceToMatch);
807         }
808     }
809 }
810 }
811
812 #endregion
813 }
814 }

```

### 1.118 ./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 ILinks<TLink> sequences, IList<TLink[]>
            ↪ groupedSequence)
13         {
14             var finalSequence = new TLink[groupedSequence.Count];
15             for (var i = 0; i < finalSequence.Length; i++)
16             {
17                 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 }

```

### 1.119 ./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)]

```

```

    set;
}

public IConverter<IList<TLink>, TLink> LinksToSequenceConverter
{
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    get;
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    set;
}

public ISequenceIndex<TLink> Index
{
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    get;
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    set;
}

public ISequenceWalker<TLink> Walker
{
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    get;
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    set;
}

public bool ReadFullSequence
{
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    get;
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    set;
}

// TODO: Реализовать компактификацию при чтении
//public bool EnforceSingleSequenceVersionOnRead { get; set; }
//public bool UseRequestMarker { get; set; }
//public bool StoreRequestResults { get; set; }

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void InitOptions(ISynchronizedLinks<TLink> links)
{
    if (UseSequenceMarker)
    {
        if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
        {
            SequenceMarkerLink = links.CreatePoint();
        }
        else
        {
            if (!links.Exists(SequenceMarkerLink))
            {
                var link = links.CreatePoint();
                if (!_equalityComparer.Equals(link, SequenceMarkerLink))
                {
                    throw new InvalidOperationException("Cannot recreate sequence marker
↪ link.");
                }
            }
        }
        if (MarkedSequenceMatcher == null)
        {
            MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
↪ SequenceMarkerLink);
        }
    }
    var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
    if (UseCompression)
    {
        if (LinksToSequenceConverter == null)
        {
            ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
            if (UseSequenceMarker)
            {
                totalSequenceSymbolFrequencyCounter = new
↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
↪ MarkedSequenceMatcher);
            }
        }
    }
}

```

```

174         else
175         {
176             totalSequenceSymbolFrequencyCounter = new
177                 ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
178         }
179         var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
180             ↪ totalSequenceSymbolFrequencyCounter);
181         var compressingConverter = new CompressingConverter<TLink>(links,
182             ↪ balancedVariantConverter, doubletFrequenciesCache);
183         LinksToSequenceConverter = compressingConverter;
184     }
185 }
186 else
187 {
188     if (LinksToSequenceConverter == null)
189     {
190         LinksToSequenceConverter = balancedVariantConverter;
191     }
192 }
193 if (UseIndex && Index == null)
194 {
195     Index = new SequenceIndex<TLink>(links);
196 }
197 if (Walker == null)
198 {
199     Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
200 }
201 }
202 }
203 }
204 }
205 }
206 }
207 }
208 }
209 }

```

#### 1.120 ./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 }

```

#### 1.121 ./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 }

```

```

20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     protected override TLink GetNextElementAfterPush(TLink element) =>
23         ↪ _links.GetTarget(element);
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     protected override IEnumerable<TLink> WalkContents(TLink element)
27     {
28         var links = _links;
29         var parts = links.GetLink(element);
30         var start = links.Constants.SourcePart;
31         for (var i = parts.Count - 1; i >= start; i--)
32         {
33             var part = parts[i];
34             if (IsElement(part))
35             {
36                 yield return part;
37             }
38         }
39     }
40 }

```

### 1.122 ./csharp/Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  //#define USEARRAYPOOL
8  #if USEARRAYPOOL
9  using Platform.Collections;
10 #endif
11
12 namespace Platform.Data.Doublets.Sequences.Walkers
13 {
14     public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
15     {
16         private static readonly EqualityComparer<TLink> _equalityComparer =
17             ↪ EqualityComparer<TLink>.Default;
18
19         private readonly Func<TLink, bool> _isElement;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
23             ↪ base(links) => _isElement = isElement;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
27             ↪ _links.IsPartialPoint;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public TLink[] ToArray(TLink sequence)
34         {
35             var length = 1;
36             var array = new TLink[length];
37             array[0] = sequence;
38             if (_isElement(sequence))
39             {
40                 return array;
41             }
42             bool hasElements;
43             do
44             {
45                 length *= 2;
46 #if USEARRAYPOOL
47                 var nextArray = ArrayPool.Allocate<ulong>(length);
48 #else
49                 var nextArray = new TLink[length];
50 #endif
51                 hasElements = false;
52                 for (var i = 0; i < array.Length; i++)
53                 {
54                     var candidate = array[i];
55                     if (_equalityComparer.Equals(array[i], default))
56                     {

```



```

54         continue;
55     }
56     var doubletOffset = i * 2;
57     if (_isElement(candidate))
58     {
59         nextArray[doubletOffset] = candidate;
60     }
61     else
62     {
63         var links = _links;
64         var link = links.GetLink(candidate);
65         var linkSource = links.GetSource(link);
66         var linkTarget = links.GetTarget(link);
67         nextArray[doubletOffset] = linkSource;
68         nextArray[doubletOffset + 1] = linkTarget;
69         if (!hasElements)
70         {
71             hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
72         }
73     }
74 }
75 #if USEARRAYPOOL
76     if (array.Length > 1)
77     {
78         ArrayPool.Free(array);
79     }
80 #endif
81     array = nextArray;
82 }
83 while (hasElements);
84 var filledElementsCount = CountFilledElements(array);
85 if (filledElementsCount == array.Length)
86 {
87     return array;
88 }
89 else
90 {
91     return CopyFilledElements(array, filledElementsCount);
92 }
93 }
94
95 [MethodImpl(MethodImplOptions.AggressiveInlining)]
96 private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
97 {
98     var finalArray = new TLink[filledElementsCount];
99     for (int i = 0, j = 0; i < array.Length; i++)
100     {
101         if (!_equalityComparer.Equals(array[i], default))
102         {
103             finalArray[j] = array[i];
104             j++;
105         }
106     }
107 #if USEARRAYPOOL
108     ArrayPool.Free(array);
109 #endif
110     return finalArray;
111 }
112
113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
114 private static int CountFilledElements(TLink[] array)
115 {
116     var count = 0;
117     for (var i = 0; i < array.Length; i++)
118     {
119         if (!_equalityComparer.Equals(array[i], default))
120         {
121             count++;
122         }
123     }
124     return count;
125 }
126 }
127 }

```

### 1.123 ./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 }

```

#### 1.124 ./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

```

```

36         while (true)
37         {
38             if (IsElement(element))
39             {
40                 if (_stack.IsEmpty)
41                 {
42                     break;
43                 }
44                 element = _stack.Pop();
45                 foreach (var output in WalkContents(element))
46                 {
47                     yield return output;
48                 }
49                 element = GetNextElementAfterPop(element);
50             }
51             else
52             {
53                 _stack.Push(element);
54                 element = GetNextElementAfterPush(element);
55             }
56         }
57     }
58 }
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected abstract TLink GetNextElementAfterPop(TLink element);
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 protected abstract TLink GetNextElementAfterPush(TLink element);
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected abstract IEnumerable<TLink> WalkContents(TLink element);
71 }
72 }

```

### 1.125 ./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);

```

```

41         _links.Update(_stack, GetStackMarker(), previousTop);
42         _links.Delete(top);
43     }
44     return element;
45 }
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
49     ↪ _links.GetOrCreate(GetTop(), element));
50 }

```

#### 1.126 ./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 }

```

#### 1.127 ./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         {

```

```

48     SyncRoot = synchronization;
49     Sync = this;
50     Unsync = links;
51     Constants = links.Constants;
52 }
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 public TLinkAddress Count(IList<TLinkAddress> restriction) =>
    ↳ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
    ↳ IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
    ↳ restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
    ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
    ↳ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
    ↳ Unsync.Update);
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 public void Delete(IList<TLinkAddress> restrictions) =>
    ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
68
69 //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
    ↳ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
70 //{
71 //    if (restriction != null && substitution != null &&
    ↳ !substitution.EqualTo(restriction))
72 //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
    ↳ substitution, substitutedHandler, Unsync.Trigger);
73
74 //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
    ↳ substitutedHandler, Unsync.Trigger);
75 //}
76 }
77 }

```

## 1.128 ./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 =
            ↳ Default<LinksConstants<ulong>>.Instance;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
21         {
22             if (sequence == null)
23             {
24                 return false;
25             }
26             var constants = links.Constants;
27             for (var i = 0; i < sequence.Length; i++)
28             {
29                 if (sequence[i] == constants.Any)
30                 {
31                     return true;
32                 }
33             }
34             return false;
35         }
36     }
37 }

```

```

36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
38     ↪ Func<Link<ulong>, bool> isElement, bool renderIndex = false, bool renderDebug =
39     ↪ false)
40 {
41     var sb = new StringBuilder();
42     var visited = new HashSet<ulong>();
43     links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
44         ↪ innerSb.Append(link.Index), renderIndex, renderDebug);
45     return sb.ToString();
46 }
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
50     ↪ Func<Link<ulong>, bool> isElement, Action<StringBuilder, Link<ulong>> appendElement,
51     ↪ bool renderIndex = false, bool renderDebug = false)
52 {
53     var sb = new StringBuilder();
54     var visited = new HashSet<ulong>();
55     links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
56         ↪ renderDebug);
57     return sb.ToString();
58 }
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
62     ↪ HashSet<ulong> visited, ulong linkIndex, Func<Link<ulong>, bool> isElement,
63     ↪ Action<StringBuilder, Link<ulong>> appendElement, bool renderIndex = false, bool
64     ↪ renderDebug = false)
65 {
66     if (sb == null)
67     {
68         throw new ArgumentNullException(nameof(sb));
69     }
70     if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
71         ↪ Constants.Itself)
72     {
73         return;
74     }
75     if (links.Exists(linkIndex))
76     {
77         if (visited.Add(linkIndex))
78         {
79             sb.Append('(');
80             var link = new Link<ulong>(links.GetLink(linkIndex));
81             if (renderIndex)
82             {
83                 sb.Append(link.Index);
84                 sb.Append(':');
85             }
86             if (link.Source == link.Index)
87             {
88                 sb.Append(link.Index);
89             }
90             else
91             {
92                 var source = new Link<ulong>(links.GetLink(link.Source));
93                 if (isElement(source))
94                 {
95                     appendElement(sb, source);
96                 }
97                 else
98                 {
99                     links.AppendStructure(sb, visited, source.Index, isElement,
100                         ↪ appendElement, renderIndex);
101                 }
102             }
103             sb.Append(' ');
104             if (link.Target == link.Index)
105             {
106                 sb.Append(link.Index);
107             }
108             else
109             {
110                 var target = new Link<ulong>(links.GetLink(link.Target));
111                 if (isElement(target))
112                 {
113

```

```

103         appendElement(sb, target);
104     }
105     else
106     {
107         links.AppendStructure(sb, visited, target.Index, isElement,
108             ↳ appendElement, renderIndex);
109     }
110     sb.Append(' ');
111 }
112 else
113 {
114     if (renderDebug)
115     {
116         sb.Append('*');
117     }
118     sb.Append(linkIndex);
119 }
120 }
121 else
122 {
123     if (renderDebug)
124     {
125         sb.Append('~');
126     }
127     sb.Append(linkIndex);
128 }
129 }
130 }
131 }

```

### 1.129 ./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

```

```

48 ///
49 ///     public ulong TransactionId
50 ///     {
51 ///         get
52 ///         {
53 ///             return (ulong) mask & TransactionIdCombined;
54 ///         }
55 ///     }
56 ///
57 ///     public UniqueTimestamp Timestamp
58 ///     {
59 ///         get
60 ///         {
61 ///             return (UniqueTimestamp)mask & TransactionIdCombined;
62 ///         }
63 ///     }
64 ///
65 ///     public TransactionItemType Type
66 ///     {
67 ///         get
68 ///         {
69 ///             // Использовать по одному биту из TransactionId и Timestamp,
70 ///             // для значения в 2 бита, которое представляет тип операции
71 ///             throw new NotImplementedException();
72 ///         }
73 ///     }
74 /// }
75 ///
76 /// private struct Transition
77 /// {
78 ///     public TransitionHeader Header;
79 ///     public Link Source;
80 ///     public Link Linker;
81 ///     public Link Target;
82 /// }
83 ///
84 /// </remarks>
85 public struct Transition : IEquatable<Transition>
86 {
87     public static readonly long Size = Structure<Transition>.Size;
88
89     public readonly ulong TransactionId;
90     public readonly Link<ulong> Before;
91     public readonly Link<ulong> After;
92     public readonly Timestamp Timestamp;
93
94     [MethodImpl(MethodImplOptions.AggressiveInlining)]
95     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
96     ↪ transactionId, Link<ulong> before, Link<ulong> after)
97     {
98         TransactionId = transactionId;
99         Before = before;
100        After = after;
101        Timestamp = uniqueTimestampFactory.Create();
102    }
103
104     [MethodImpl(MethodImplOptions.AggressiveInlining)]
105     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
106     ↪ transactionId, Link<ulong> before) : this(uniqueTimestampFactory, transactionId,
107     ↪ before, default) { }
108
109     [MethodImpl(MethodImplOptions.AggressiveInlining)]
110     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
111     ↪ transactionId) : this(uniqueTimestampFactory, transactionId, default, default) {
112     ↪ }
113
114     [MethodImpl(MethodImplOptions.AggressiveInlining)]
115     public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
116     ↪ {After}";
117
118     [MethodImpl(MethodImplOptions.AggressiveInlining)]
119     public override bool Equals(object obj) => obj is Transition transition ?
120     ↪ Equals(transition) : false;
121
122     [MethodImpl(MethodImplOptions.AggressiveInlining)]
123     public override int GetHashCode() => (TransactionId, Before, After,
124     ↪ Timestamp).GetHashCode();

```



```

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     }
188 }

```

```

187         _layer._lastCommittedTransactionId = _layer._currentTransactionId;
188         IsCommitted = true;
189     }
190
191     [MethodImpl(MethodImplOptions.AggressiveInlining)]
192     private void Revert()
193     {
194         EnsureTransactionAllowsWriteOperations(this);
195         var transitionsToRevert = new Transition[_transitions.Count];
196         _transitions.CopyTo(transitionsToRevert, 0);
197         for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
198         {
199             _layer.RevertTransition(transitionsToRevert[i]);
200         }
201         IsReverted = true;
202     }
203
204     [MethodImpl(MethodImplOptions.AggressiveInlining)]
205     public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
206     ↪ Transaction transaction)
207     {
208         layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
209         layer._currentTransactionTransitions = transaction._transitions;
210         layer._currentTransaction = transaction;
211     }
212
213     [MethodImpl(MethodImplOptions.AggressiveInlining)]
214     public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
215     {
216         if (transaction.IsReverted)
217         {
218             throw new InvalidOperationException("Transation is reverted.");
219         }
220         if (transaction.IsCommitted)
221         {
222             throw new InvalidOperationException("Transation is committed.");
223         }
224     }
225
226     [MethodImpl(MethodImplOptions.AggressiveInlining)]
227     protected override void Dispose(bool manual, bool wasDisposed)
228     {
229         if (!wasDisposed && _layer != null && !_layer.Disposable.IsDisposed)
230         {
231             if (!IsCommitted && !IsReverted)
232             {
233                 Revert();
234             }
235             _layer.ResetCurrentTransation();
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 закомиченную 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
        ↳ supported yet.");
270     }
271     if (lastCommittedTransition == default)
272     {
273         FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
274     }
275     _lastCommittedTransition = lastCommittedTransition;
276     // TODO: Think about a better way to calculate or store this value
277     var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
278     _lastCommittedTransactionId = allTransitions.Length > 0 ? allTransitions.Max(x =>
        ↳ x.TransactionId) : 0;
279     _uniqueTimestampFactory = new UniqueTimestampFactory();
280     _logAddress = logAddress;
281     _log = FileHelpers.Append(logAddress);
282     _transitions = new Queue<Transition>();
283     _transitionsPusher = new Task(TransitionsPusher);
284     _transitionsPusher.Start();
285 }
286
287 [MethodImpl(MethodImplOptions.AggressiveInlining)]
288 public IList<ulong> GetLinkValue(ulong link) => _links.GetLink(link);
289
290 [MethodImpl(MethodImplOptions.AggressiveInlining)]
291 public override ulong Create(IList<ulong> restrictions)
292 {
293     var createdLinkIndex = _links.Create();
294     var createdLink = new Link<ulong>(_links.GetLink(createdLinkIndex));
295     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
        ↳ default, createdLink));
296     return createdLinkIndex;
297 }
298
299 [MethodImpl(MethodImplOptions.AggressiveInlining)]
300 public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
301 {
302     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,
        ↳ beforeLink, afterLink));
307     return linkIndex;
308 }
309
310 [MethodImpl(MethodImplOptions.AggressiveInlining)]
311 public override void Delete(IList<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,
        ↳ deletedLink, default));
317 }
318
319 [MethodImpl(MethodImplOptions.AggressiveInlining)]
320 private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
    ↳ _transitions;
321
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
333 [MethodImpl(MethodImplOptions.AggressiveInlining)]
334 private void RevertTransition(Transition transition)
335 {
336     if (transition.After.IsNull()) // Revert Deletion with Creation

```

```

337     {
338         _links.Create();
339     }
340     else if (transition.Before.IsNull()) // Revert Creation with Deletion
341     {
342         _links.Delete(transition.After.Index);
343     }
344     else // Revert Update
345     {
346         _links.Update(new[] { transition.After.Index, transition.Before.Source,
            ↪ transition.Before.Target });
347     }
348 }
349
350 [MethodImpl(MethodImplOptions.AggressiveInlining)]
351 private void ResetCurrentTransation()
352 {
353     _currentTransactionId = 0;
354     _currentTransactionTransitions = null;
355     _currentTransaction = null;
356 }
357
358 [MethodImpl(MethodImplOptions.AggressiveInlining)]
359 private void PushTransitions()
360 {
361     if (_log == null || _transitions == null)
362     {
363         return;
364     }
365     for (var i = 0; i < _transitions.Count; i++)
366     {
367         var transition = _transitions.Dequeue();
368
369         _log.Write(transition);
370         _lastCommittedTransition = transition;
371     }
372 }
373
374 [MethodImpl(MethodImplOptions.AggressiveInlining)]
375 private void TransitionsPusher()
376 {
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 }

```

### 1.130 ./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             {
29                 var unaryNumber =
30                     ↪ _addressToNumberConverter.Convert(_charToAddressConverter.Convert(source));
31                 return _links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
32             }
33         }
34     }
35 }

```

### 1.131 ./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>,
11         ↪ IConverter<string, TLink>
12     {
13         private readonly IConverter<string, IList<TLink>> _stringToUnicodeSymbolListConverter;
14         private readonly IConverter<IList<TLink>, TLink> _unicodeSymbolListToSequenceConverter;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
18             ↪ IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
19             ↪ unicodeSymbolListToSequenceConverter) : base(links)
20         {
21             _stringToUnicodeSymbolListConverter = stringToUnicodeSymbolListConverter;
22             _unicodeSymbolListToSequenceConverter = unicodeSymbolListToSequenceConverter;
23         }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
27             ↪ IList<TLink>> stringToUnicodeSymbolListConverter, ISequenceIndex<TLink> index,
28             ↪ IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
29             ↪ unicodeSequenceMarker)
30         : this(links, stringToUnicodeSymbolListConverter, new
31             ↪ UnicodeSymbolsListToUnicodeSequenceConverter<TLink>(links, index,
32             ↪ listToSequenceLinkConverter, unicodeSequenceMarker)) { }
33     }
34 }

```

```

25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
    ↪ charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
    ↪ TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker)
28         : this(links, new
    ↪ StringToUnicodeSymbolsListConverter<TLink>(charToUnicodeSymbolConverter), index,
    ↪ listToSequenceLinkConverter, unicodeSequenceMarker) { }
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
    ↪ charToUnicodeSymbolConverter, IConverter<IList<TLink>, TLink>
    ↪ listToSequenceLinkConverter, TLink unicodeSequenceMarker)
32         : this(links, charToUnicodeSymbolConverter, new Unindex<TLink>(),
    ↪ listToSequenceLinkConverter, unicodeSequenceMarker) { }
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
    ↪ IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
    ↪ listToSequenceLinkConverter, TLink unicodeSequenceMarker)
36         : this(links, stringToUnicodeSymbolListConverter, new Unindex<TLink>(),
    ↪ listToSequenceLinkConverter, unicodeSequenceMarker) { }
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     public TLink Convert(string source)
40     {
41         var elements = _stringToUnicodeSymbolListConverter.Convert(source);
42         return _unicodeSymbolListToSequenceConverter.Convert(elements);
43     }
44 }
45 }

```

### 1.132 ./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSymbolsListConverter.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.Unicode
8  {
9      public class StringToUnicodeSymbolsListConverter<TLink> : IConverter<string, IList<TLink>>
10     {
11         private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public StringToUnicodeSymbolsListConverter(IConverter<char, TLink>
    ↪ charToUnicodeSymbolConverter) => _charToUnicodeSymbolConverter =
    ↪ charToUnicodeSymbolConverter;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public IList<TLink> Convert(string source)
18         {
19             var elements = new TLink[source.Length];
20             for (var i = 0; i < elements.Length; i++)
21             {
22                 elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
23             }
24             return elements;
25         }
26     }
27 }

```

### 1.133 ./csharp/Platform.Data.Doublets/Unicode/UnicodeMap.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Globalization;
4  using System.Runtime.CompilerServices;
5  using System.Text;
6  using Platform.Data.Sequences;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Unicode
11 {
12     public class UnicodeMap
13     {
14         public static readonly ulong FirstCharLink = 1;
15         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;

```

```

16 public static readonly ulong MapSize = 1+ char.MaxValue;
17
18 private readonly ILinks<ulong> _links;
19 private bool _initialized;
20
21 [MethodImpl(MethodImplOptions.AggressiveInlining)]
22 public UnicodeMap(ILinks<ulong> links) => _links = links;
23
24 [MethodImpl(MethodImplOptions.AggressiveInlining)]
25 public static UnicodeMap InitNew(ILinks<ulong> links)
26 {
27     var map = new UnicodeMap(links);
28     map.Init();
29     return map;
30 }
31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 public void Init()
34 {
35     if (_initialized)
36     {
37         return;
38     }
39     _initialized = true;
40     var firstLink = _links.CreatePoint();
41     if (firstLink != FirstCharLink)
42     {
43         _links.Delete(firstLink);
44     }
45     else
46     {
47         for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
48         {
49             // From NIL to It (NIL -> Character) transformation meaning, (or infinite
50             // ↪ amount of NIL characters before actual Character)
51             var createdLink = _links.CreatePoint();
52             _links.Update(createdLink, firstLink, createdLink);
53             if (createdLink != i)
54             {
55                 throw new InvalidOperationException("Unable to initialize UTF 16
56                 ↪ table.");
57             }
58         }
59     }
60 }
61
62 // 0 - null link
63 // 1 - nil character (0 character)
64 // ...
65 // 65536 (0(1) + 65535 = 65536 possible values)
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 public static ulong FromCharToLink(char character) => (ulong)character + 1;
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 public static char FromLinkToChar(ulong link) => (char)(link - 1);
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 public static bool IsCharLink(ulong link) => link <= MapSize;
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 public static string FromLinksToString(IList<ulong> linksList)
78 {
79     var sb = new StringBuilder();
80     for (int i = 0; i < linksList.Count; i++)
81     {
82         sb.Append(FromLinkToChar(linksList[i]));
83     }
84     return sb.ToString();
85 }
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
89 {
90     var sb = new StringBuilder();
91     if (links.Exists(link))
92     {
93         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     return sb.ToString();
99 }
100
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
103     ↪ chars.Length);
104
105 [MethodImpl(MethodImplOptions.AggressiveInlining)]
106 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
107 {
108     // char array to ulong array
109     var linksSequence = new ulong[count];
110     for (var i = 0; i < count; i++)
111     {
112         linksSequence[i] = FromCharToLink(chars[i]);
113     }
114     return linksSequence;
115 }
116
117 [MethodImpl(MethodImplOptions.AggressiveInlining)]
118 public static ulong[] FromStringToLinkArray(string sequence)
119 {
120     // char array to ulong array
121     var linksSequence = new ulong[sequence.Length];
122     for (var i = 0; i < sequence.Length; i++)
123     {
124         linksSequence[i] = FromCharToLink(sequence[i]);
125     }
126     return linksSequence;
127 }
128
129 [MethodImpl(MethodImplOptions.AggressiveInlining)]
130 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
131 {
132     var result = new List<ulong[]>();
133     var offset = 0;
134     while (offset < sequence.Length)
135     {
136         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
137         var relativeLength = 1;
138         var absoluteLength = offset + relativeLength;
139         while (absoluteLength < sequence.Length &&
140             currentCategory ==
141             ↪ CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
142         {
143             relativeLength++;
144             absoluteLength++;
145         }
146         // char array to ulong array
147         var innerSequence = new ulong[relativeLength];
148         var maxLength = offset + relativeLength;
149         for (var i = offset; i < maxLength; i++)
150         {
151             innerSequence[i - offset] = FromCharToLink(sequence[i]);
152         }
153         result.Add(innerSequence);
154         offset += relativeLength;
155     }
156     return result;
157 }
158
159 [MethodImpl(MethodImplOptions.AggressiveInlining)]
160 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
161 {
162     var result = new List<ulong[]>();
163     var offset = 0;
164     while (offset < array.Length)
165     {
166         var relativeLength = 1;
167         if (array[offset] <= LastCharLink)
168         {

```



```

167         var currentCategory =
168             ↳ CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
169         var absoluteLength = offset + relativeLength;
170         while (absoluteLength < array.Length &&
171             array[absoluteLength] <= LastCharLink &&
172             currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
173                 ↳ array[absoluteLength])))
174         {
175             relativeLength++;
176             absoluteLength++;
177         }
178     }
179     else
180     {
181         var absoluteLength = offset + relativeLength;
182         while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
183         {
184             relativeLength++;
185             absoluteLength++;
186         }
187         // copy array
188         var innerSequence = new ulong[relativeLength];
189         var maxLength = offset + relativeLength;
190         for (var i = offset; i < maxLength; i++)
191         {
192             innerSequence[i - offset] = array[i];
193         }
194         result.Add(innerSequence);
195         offset += relativeLength;
196     }
197     return result;
198 }
199 }

```

#### 1.134 ./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs

```

1 using System;
2 using System.Runtime.CompilerServices;
3 using Platform.Interfaces;
4 using Platform.Converters;
5 using Platform.Data.Doublets.Sequences.Walkers;
6 using System.Text;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Unicode
11 {
12     public class UnicodeSequenceToStringConverter<TLink> : LinksOperatorBase<TLink>,
13         ↳ IConverter<TLink, string>
14     {
15         private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
16         private readonly ISequenceWalker<TLink> _sequenceWalker;
17         private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
21             ↳ unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
22             ↳ IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
23         {
24             _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
25             _sequenceWalker = sequenceWalker;
26             _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public string Convert(TLink source)
31         {
32             if (!_unicodeSequenceCriterionMatcher.IsMatched(source))
33             {
34                 throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
35                     ↳ not a unicode sequence.");
36             }
37             var sequence = _links.GetSource(source);
38             var sb = new StringBuilder();
39             foreach (var character in _sequenceWalker.Walk(sequence))
40             {
41                 sb.Append(_unicodeSymbolToCharConverter.Convert(character));
42             }
43             return sb.ToString();
44         }
45     }
46 }

```

```

40     }
41 }
42 }

```

### 1.135 ./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs

```

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

```

### 1.136 ./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.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 UnicodeSymbolsListToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
11         ↪ IConverter<IList<TLink>, TLink>
12     {
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 UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
19             ↪ ISequenceIndex<TLink> index, IConverter<IList<TLink>, TLink>
20             ↪ listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
21         {
22             _index = index;
23             _listToSequenceLinkConverter = listToSequenceLinkConverter;
24             _unicodeSequenceMarker = unicodeSequenceMarker;
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
29             ↪ IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
30             ↪ unicodeSequenceMarker)
31             : this(links, new Unindex<TLink>(), listToSequenceLinkConverter,
32                 ↪ unicodeSequenceMarker) { }
33     }
34 }

```

```

28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     public TLink Convert(IList<TLink> list)
30     {
31         _index.Add(list);
32         var sequence = _listToSequenceLinkConverter.Convert(list);
33         return _links.GetOrCreate(sequence, _unicodeSequenceMarker);
34     }
35 }
36 }

```

### 1.137 ./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 }

```

### 1.138 ./csharp/Platform.Data.Doublets.Tests/ILinksExtensionsTests.cs

```

1  using Xunit;
2
3  namespace Platform.Data.Doublets.Tests
4  {
5     public class ILinksExtensionsTests
6     {
7         [Fact]
8         public void FormatTest()
9         {
10             using (var scope = new TempLinksTestScope())
11             {
12                 var links = scope.Links;

```

```

13         var link = links.Create();
14         var linkString = links.Format(link);
15         Assert.Equal("(1: 1 1)", linkString);
16     }
17 }
18 }
19 }

```

### 1.139 ./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 }

```

### 1.140 ./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 using Platform.Data.Doublets.Memory;
21
22 namespace Platform.Data.Doublets.Tests
23 {
24     public static class OptimalVariantSequenceTests
25     {
26         private static readonly string _sequenceExample = "зеленела зелёная зелень";
27         private static readonly string _loremIpsumExample = @"Lorem ipsum dolor sit amet,
28             ↪ consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore
29             ↪ magna aliqua.
30 Facilisi nullam vehicula ipsum a arcu cursus vitae congue mauris.
31 Et malesuada fames ac turpis egestas sed.
32 Eget velit aliquet sagittis id consectetur purus.
33 Dignissim cras tincidunt lobortis feugiat vivamus.
34 Vitae aliquet nec ullamcorper sit.
35 Lectus quam id leo in vitae.
36 Tortor dignissim convallis aenean et tortor at risus viverra adipiscing.
37 Sed risus ultricies tristique nulla aliquet enim tortor at auctor.
38 Integer eget aliquet nibh praesent tristique.
39 Vitae congue eu consequat ac felis donec et odio.
40 Tristique et egestas quis ipsum suspendisse.
41 Suspendisse potenti nullam ac tortor vitae purus faucibus ornare.
42 Nulla facilisi etiam dignissim diam quis enim lobortis scelerisque.
43 Imperdiet proin fermentum leo vel orci.
44 In ante metus dictum at tempor commodo.
45 Nisi lacus sed viverra tellus in.
46 Quam vulputate dignissim suspendisse in.
47 Elit scelerisque mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus.
48 Gravidia cum sociis natoque penatibus et magnis dis parturient.
49 Risus quis varius quam quisque id diam.
50 Congue nisi vitae suscipit tellus mauris a diam maecenas.

```

```

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

```

```

117     links.UseUnicode();
118
119     var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
120
121     var totalSequenceSymbolFrequencyCounter = new
122     ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
123
124     var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
125     ↪ totalSequenceSymbolFrequencyCounter);
126
127     var index = new
128     ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
129     var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
130
131     var sequenceToItsLocalElementLevelsConverter = new
132     ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
133     ↪ linkToItsFrequencyNumberConverter);
134     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
135     ↪ sequenceToItsLocalElementLevelsConverter);
136
137     var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
138     ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
139
140     ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
141     ↪ index, optimalVariantConverter);
142 }
143
144 private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
145 ↪ SequenceToItsLocalElementLevelsConverter<ulong>
146 ↪ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
147 ↪ OptimalVariantConverter<ulong> optimalVariantConverter)
148 {
149     index.Add(sequence);
150
151     var optimalVariant = optimalVariantConverter.Convert(sequence);
152
153     var readSequence1 = sequences.ToList(optimalVariant);
154
155     Assert.True(sequence.SequenceEqual(readSequence1));
156 }
157
158 [Fact]
159 public static void SavedSequencesOptimizationTest()
160 {
161     LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
162     ↪ (long.MaxValue + 1UL, ulong.MaxValue));
163
164     using (var memory = new HeapResizableDirectMemory())
165     using (var disposableLinks = new UInt64UnitedMemoryLinks(memory,
166     ↪ UInt64UnitedMemoryLinks.DefaultLinksSizeStep, constants, IndexTreeType.Default))
167     {
168         var links = new UInt64Links(disposableLinks);
169
170         var root = links.CreatePoint();
171
172         //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
173         var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
174
175         var unicodeSymbolMarker = links.GetOrCreate(root,
176     ↪ addressToNumberConverter.Convert(1));
177         var unicodeSequenceMarker = links.GetOrCreate(root,
178     ↪ addressToNumberConverter.Convert(2));
179
180         var totalSequenceSymbolFrequencyCounter = new
181     ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
182         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
183     ↪ totalSequenceSymbolFrequencyCounter);
184         var index = new
185     ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
186         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
187         var sequenceToItsLocalElementLevelsConverter = new
188     ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
189     ↪ linkToItsFrequencyNumberConverter);
190         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
191     ↪ sequenceToItsLocalElementLevelsConverter);

```

```

172     var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
173     ↪ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
174
175     var unicodeSequencesOptions = new SequencesOptions<ulong>()
176     {
177         UseSequenceMarker = true,
178         SequenceMarkerLink = unicodeSequenceMarker,
179         UseIndex = true,
180         Index = index,
181         LinksToSequenceConverter = optimalVariantConverter,
182         Walker = walker,
183         UseGarbageCollection = true
184     };
185
186     var unicodeSequences = new Sequences.Sequences(new
187     ↪ SynchronizedLinks<ulong>(links), unicodeSequencesOptions);
188
189     // Create some sequences
190     var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
191     ↪ StringSplitOptions.RemoveEmptyEntries);
192     var arrays = strings.Select(x => x.Select(y =>
193     ↪ addressToNumberConverter.Convert(y)).ToArray()).ToArray();
194     for (int i = 0; i < arrays.Length; i++)
195     {
196         unicodeSequences.Create(arrays[i].ShiftRight());
197     }
198
199     var linksCountAfterCreation = links.Count();
200
201     // get list of sequences links
202     // for each sequence link
203     // create new sequence version
204     // if new sequence is not the same as sequence link
205     // delete sequence link
206     // collect garbadge
207     unicodeSequences.CompactAll();
208
209     var linksCountAfterCompactification = links.Count();
210
211     Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
212 }

```

#### 1.141 ./csharp/Platform.Data.Doublets.Tests/ReadSequenceTests.cs

```

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

```

```

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

```

#### 1.142 ./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                 {

```



```

43     using (var memory = new
        ↳ HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
44     using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
        ↳ UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
45     {
46         memoryAdapter.TestNonexistentReferences();
47     }
48 }
49
50 private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
51 {
52     var link = memoryAdapter.Create();
53     memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
54     var resultLink = _constants.Null;
55     memoryAdapter.Each(foundLink =>
56     {
57         resultLink = foundLink[_constants.IndexPart];
58         return _constants.Break;
59     }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
60     Assert.True(resultLink == link);
61     Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
62     memoryAdapter.Delete(link);
63 }
64 }
65 }

```

#### 1.143 ./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,
                ↳ UnitedMemoryLinks<ulong>>>())
50             {
51                 var links = scope.Use<ILinks<ulong>>();

```

```

52         Assert.IsType<UnitedMemoryLinks<ulong>>(links);
53     }
54 }
55 }
56 }

```

#### 1.144 ./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))

```

```

73 // using (var links = new Links(memoryAdapter))
74 // {
75 //     var sequence = new ulong[sequenceLength];
76 //     for (var i = 0; i < sequenceLength; i++)
77 //         sequence[i] = links.Create(itself, itself);
78
79 //     SequencesOptions o = new SequencesOptions();
80
81 // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
82 //     o.
83
84 //     var sequences = new Sequences(links);
85
86 //     var sw1 = Stopwatch.StartNew();
87 //     var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
88
89 //     var sw2 = Stopwatch.StartNew();
90 //     var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
91
92 //     Assert.True(results1.Count > results2.Length);
93 //     Assert.True(sw1.Elapsed > sw2.Elapsed);
94
95 //     for (var i = 0; i < sequenceLength; i++)
96 //         links.Delete(sequence[i]);
97 // }
98
99 // File.Delete(tempFilename);
100 //}
101
102 [Fact]
103 public static void AllVariantsSearchTest()
104 {
105     const long sequenceLength = 8;
106
107     using (var scope = new TempLinksTestScope(useSequences: true))
108     {
109         var links = scope.Links;
110         var sequences = scope.Sequences;
111
112         var sequence = new ulong[sequenceLength];
113         for (var i = 0; i < sequenceLength; i++)
114         {
115             sequence[i] = links.Create();
116         }
117
118         var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
119
120         //for (int i = 0; i < createResults.Length; i++)
121         //    sequences.Create(createResults[i]);
122
123         var sw0 = Stopwatch.StartNew();
124         var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
125
126         var sw1 = Stopwatch.StartNew();
127         var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
128
129         var sw2 = Stopwatch.StartNew();
130         var searchResults2 = sequences.Each1(sequence); sw2.Stop();
131
132         var sw3 = Stopwatch.StartNew();
133         var searchResults3 = sequences.Each(sequence.ShiftRight()); sw3.Stop();
134
135         var intersection0 = createResults.Intersect(searchResults0).ToList();
136         Assert.True(intersection0.Count == searchResults0.Count);
137         Assert.True(intersection0.Count == createResults.Length);
138
139         var intersection1 = createResults.Intersect(searchResults1).ToList();
140         Assert.True(intersection1.Count == searchResults1.Count);
141         Assert.True(intersection1.Count == createResults.Length);
142
143         var intersection2 = createResults.Intersect(searchResults2).ToList();
144         Assert.True(intersection2.Count == searchResults2.Count);
145         Assert.True(intersection2.Count == createResults.Length);
146
147         var intersection3 = createResults.Intersect(searchResults3).ToList();
148         Assert.True(intersection3.Count == searchResults3.Count);
149         Assert.True(intersection3.Count == createResults.Length);
150
151         for (var i = 0; i < sequenceLength; i++)
152         {

```

```

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

```

```

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

```

```

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

```

```
379 /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%  
→ 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  
→ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
```

```
380 private static readonly string _exampleText =  
381     @"([english  
→     version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
```

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

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

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

388 `[[чёрное пространство, чёрная`  
389 `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 А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть  
391 → так? Инверсия? Отражение? Сумма?

392 `[[белая точка, чёрная`  
393 `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 А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет  
395 → если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой?  
→ Гранью? Разделителем? Единицей?

396 `[[две белые точки, чёрная вертикальная`  
397 `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 Можно ли замкнуть движение? Может ли это быть кругом? Можно ли замкнуть время? Или остаётся  
399 → 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 `[[белая вертикальная линия, чёрный`  
401 `круг](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png ""белая`  
→ `вертикальная линия, чёрный`  
→ `круг""] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png)`

402 Как ещё можно использовать грань, черту, линию? А что если она может что-то соединять, может  
403 → 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 `[[белый круг, чёрная горизонтальная`  
405 `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 Соединять, допустим, а какой смысл в этом есть ещё? Что если помимо смысла ""соединить,  
407 → связать"", есть ещё и смысл направления ""от начала к концу""? От предка к потомку? От  
→ родителя к ребёнку? От общего к частному?

408 `[[белая горизонтальная линия, чёрная горизонтальная`  
409 `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 Шаг назад. Возьмём опять отделённую область, которая лишь та же замкнутая линия, что ещё она  
411 → 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?

```

413  [![белая связь, чёрная направленная
    ↳ связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png "белая
    ↳ связь, чёрная направленная
    ↳ связь")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png)
414
415  Допустим у нас есть смысл ""связать"" и смысл ""направления"", много ли это нам даёт? Много ли
    ↳ вариантов интерпретации? А что если уточнить, каким именно образом выполнена связь? Что если
    ↳ можно задать ей чёткий, конкретный смысл? Что это будет? Тип? Глагол? Связка? Действие?
    ↳ Трансформация? Переход из состояния в состояние? Или всё это и есть объект, суть которого в
    ↳ его конечном состоянии, если конечно конец определён направлением?
416
417  [![белая обычная и направленная связи, чёрная типизированная
    ↳ связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png "белая
    ↳ обычная и направленная связи, чёрная типизированная
    ↳ связь")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png)
418
419  А что если всё это время, мы смотрели на суть как бы снаружи? Можно ли взглянуть на это изнутри?
    ↳ Что будет внутри объектов? Объекты ли это? Или это связи? Может ли эта структура описать
    ↳ сама себя? Но что тогда получится, разве это не рекурсия? Может это фрактал?
420
421  [![белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная
    ↳ связь с рекурсивной внутренней
    ↳ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png
    ↳ ""белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с рекурсивной внутренней структурой"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png)
422
423  На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
    ↳ рекурсии или фрактала?
424
425  [![белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с двойной рекурсивной внутренней
    ↳ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png
    ↳ ""белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с двойной рекурсивной внутренней структурой"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)
426
427  Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
    ↳ Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
428
429  [![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности,
    ↳ чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://
    ↳ raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png "белая обычная и
    ↳ направленная связи со структурой из 8 цветных элементов последовательности, чёрная
    ↳ типизированная связь со структурой из 8 цветных элементов последовательности"")](https://raw
    ↳ .githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)
430
431  ...
432
433  [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-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,
458         ↪ totalSequenceSymbolFrequencyCounter);
459     var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
460         ↪ balancedVariantConverter, doubletFrequenciesCache);
461
462     var compressedVariant = compressingConverter.Convert(sequence);
463
464     // 1: [1]          (1->1) point
465     // 2: [2]          (2->2) point
466     // 3: [1,2]        (1->2) doublet
467     // 4: [1,2,1,2]    (3->3) doublet
468
469     Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
470     Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
471     Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
472     Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
473
474     var source = _constants.SourcePart;
475     var target = _constants.TargetPart;
476
477     Assert.True(links.GetByKey(compressedVariant, source, source) == sequence[0]);
478     Assert.True(links.GetByKey(compressedVariant, source, target) == sequence[1]);
479     Assert.True(links.GetByKey(compressedVariant, target, source) == sequence[2]);
480     Assert.True(links.GetByKey(compressedVariant, target, target) == sequence[3]);
481
482     // 4 - length of sequence
483     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
484         ↪ == sequence[0]);
485     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
486         ↪ == sequence[1]);
487     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
488         ↪ == sequence[2]);
489     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
490         ↪ == sequence[3]);
491 }
492
493 [Fact]
494 public static void CompressionEfficiencyTest()
495 {
496     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
497         ↪ StringSplitOptions.RemoveEmptyEntries);
498     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
499     var totalCharacters = arrays.Select(x => x.Length).Sum();
500
501     using (var scope1 = new TempLinksTestScope(useSequences: true))
502     using (var scope2 = new TempLinksTestScope(useSequences: true))
503     using (var scope3 = new TempLinksTestScope(useSequences: true))
504     {
505         scope1.Links.Unsync.UseUnicode();
506         scope2.Links.Unsync.UseUnicode();
507         scope3.Links.Unsync.UseUnicode();
508
509         var balancedVariantConverter1 = new
510             ↪ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
511         var totalSequenceSymbolFrequencyCounter = new
512             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
513         var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
514             ↪ totalSequenceSymbolFrequencyCounter);
515         var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
516             ↪ balancedVariantConverter1, linkFrequenciesCache1,
517             ↪ doInitialFrequenciesIncrement: false);
518
519         //var compressor2 = scope2.Sequences;
520         var compressor3 = scope3.Sequences;
521
522         var constants = Default<LinksConstants<ulong>>.Instance;
523
524         var sequences = compressor3;
525         //var meaningRoot = links.CreatePoint();
526         //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
527         //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
528         //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
529             ↪ constants.Itself);
530
531         //var unaryNumberToAddressConverter = new
532             ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);

```

```

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

```

```

585 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
    ↳ Optimal variant: {elapsed3}");
586
587 // Assert.True(elapsed1 > elapsed2);
588
589 // Checks
590 for (int i = START; i < END; i++)
591 {
592     var sequence1 = compressed1[i];
593     var sequence2 = compressed2[i];
594     var sequence3 = compressed3[i];
595
596     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
    ↳ scope1.Links.Unsync);
597
598     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
    ↳ scope2.Links.Unsync);
599
600     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
    ↳ scope3.Links.Unsync);
601
602     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
    ↳ link.IsPartialPoint());
603     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
    ↳ link.IsPartialPoint());
604     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
    ↳ link.IsPartialPoint());
605
606     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
    ↳ arrays[i].Length > 3)
607     //    Assert.False(structure1 == structure2);
608     //if (sequence2 != Constants.Null && sequence3 != Constants.Null &&
    ↳ arrays[i].Length > 3)
609     //    Assert.False(structure3 == structure2);
610
611     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
612     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
613 }
614
615 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
    ↳ totalCharacters);
616 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
    ↳ totalCharacters);
617 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
    ↳ totalCharacters);
618
619 Console.WriteLine($"{{(double)(scope1.Links.Unsync.Count() - initialCount1) /
    ↳ totalCharacters}} | {{(double)(scope2.Links.Unsync.Count() - initialCount2) /
    ↳ totalCharacters}} | {{(double)(scope3.Links.Unsync.Count() - initialCount3) /
    ↳ totalCharacters}}");
620
621 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
    ↳ scope2.Links.Unsync.Count() - initialCount2);
622 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
    ↳ scope2.Links.Unsync.Count() - initialCount2);
623
624 var duplicateProvider1 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
625 var duplicateProvider2 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
626 var duplicateProvider3 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
627
628 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
629 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
630 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
631
632 var duplicates1 = duplicateCounter1.Count();
633
634 ConsoleHelpers.Debug("-----");
635
636 var duplicates2 = duplicateCounter2.Count();
637
638 ConsoleHelpers.Debug("-----");
639
640 var duplicates3 = duplicateCounter3.Count();
641
642 Console.WriteLine($"{{duplicates1}} | {{duplicates2}} | {{duplicates3}}");

```

```

643         linkFrequenciesCache1.ValidateFrequencies();
644         linkFrequenciesCache3.ValidateFrequencies();
645     }
646 }
647
648 [Fact]
649 public static void CompressionStabilityTest()
650 {
651     // TODO: Fix bug (do a separate test)
652     //const ulong minNumbers = 0;
653     //const ulong maxNumbers = 1000;
654
655     const ulong minNumbers = 10000;
656     const ulong maxNumbers = 12500;
657
658     var strings = new List<string>();
659
660     for (ulong i = minNumbers; i < maxNumbers; i++)
661     {
662         strings.Add(i.ToString());
663     }
664
665     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
666     var totalCharacters = arrays.Select(x => x.Length).Sum();
667
668     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
669         ↪ SequencesOptions<ulong> { UseCompression = true,
670         ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
671     using (var scope2 = new TempLinksTestScope(useSequences: true))
672     {
673         scope1.Links.UseUnicode();
674         scope2.Links.UseUnicode();
675
676         //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
677         var compressor1 = scope1.Sequences;
678         var compressor2 = scope2.Sequences;
679
680         var compressed1 = new ulong[arrays.Length];
681         var compressed2 = new ulong[arrays.Length];
682
683         var sw1 = Stopwatch.StartNew();
684
685         var START = 0;
686         var END = arrays.Length;
687
688         // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
689         // Stability issue starts at 10001 or 11000
690         //for (int i = START; i < END; i++)
691         //{
692         //    var first = compressor1.Compress(arrays[i]);
693         //    var second = compressor1.Compress(arrays[i]);
694
695         //    if (first == second)
696         //        compressed1[i] = first;
697         //    else
698         //    {
699         //        // TODO: Find a solution for this case
700         //    }
701         //}
702
703         for (int i = START; i < END; i++)
704         {
705             var first = compressor1.Create(arrays[i].ShiftRight());
706             var second = compressor1.Create(arrays[i].ShiftRight());
707
708             if (first == second)
709             {
710                 compressed1[i] = first;
711             }
712             else
713             {
714                 // TODO: Find a solution for this case
715             }
716         }
717
718         var elapsed1 = sw1.Elapsed;
719
720         var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);

```

```

721     var sw2 = Stopwatch.StartNew();
722
723     for (int i = START; i < END; i++)
724     {
725         var first = balancedVariantConverter.Convert(arrays[i]);
726         var second = balancedVariantConverter.Convert(arrays[i]);
727
728         if (first == second)
729         {
730             compressed2[i] = first;
731         }
732     }
733
734     var elapsed2 = sw2.Elapsed;
735
736     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
737     ↪ {elapsed2}");
738
739     Assert.True(elapsed1 > elapsed2);
740
741     // Checks
742     for (int i = START; i < END; i++)
743     {
744         var sequence1 = compressed1[i];
745         var sequence2 = compressed2[i];
746
747         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
748         {
749             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
750             ↪ scope1.Links);
751
752             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
753             ↪ scope2.Links);
754
755             //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
756             ↪ link.IsPartialPoint());
757             //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
758             ↪ link.IsPartialPoint());
759
760             //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
761             ↪ arrays[i].Length > 3)
762             //    Assert.False(structure1 == structure2);
763
764             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
765         }
766     }
767
768     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
769     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
770
771     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
772     ↪ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
773     ↪ totalCharacters}}");
774
775     Assert.True(scope1.Links.Count() <= scope2.Links.Count());
776
777     //compressor1.ValidateFrequencies();
778 }
779
780 [Fact]
781 public static void RandomNumbersCompressionQualityTest()
782 {
783     const ulong N = 500;
784
785     //const ulong minNumbers = 10000;
786     //const ulong maxNumbers = 20000;
787
788     //var strings = new List<string>();
789
790     //for (ulong i = 0; i < N; i++)
791     //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
792     ↪ maxNumbers).ToString());
793
794     var strings = new List<string>();
795
796     for (ulong i = 0; i < N; i++)
797     {
798         strings.Add(RandomHelpers.Default.NextUInt64().ToString());
799     }

```

```

791 }
792
793 strings = strings.Distinct().ToList();
794
795 var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
796 var totalCharacters = arrays.Select(x => x.Length).Sum();
797
798 using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
    ↳ SequencesOptions<ulong> { UseCompression = true,
    ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
799 using (var scope2 = new TempLinksTestScope(useSequences: true))
800 {
801     scope1.Links.UseUnicode();
802     scope2.Links.UseUnicode();
803
804     var compressor1 = scope1.Sequences;
805     var compressor2 = scope2.Sequences;
806
807     var compressed1 = new ulong[arrays.Length];
808     var compressed2 = new ulong[arrays.Length];
809
810     var sw1 = Stopwatch.StartNew();
811
812     var START = 0;
813     var END = arrays.Length;
814
815     for (int i = START; i < END; i++)
816     {
817         compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
818     }
819
820     var elapsed1 = sw1.Elapsed;
821
822     var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
823
824     var sw2 = Stopwatch.StartNew();
825
826     for (int i = START; i < END; i++)
827     {
828         compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
829     }
830
831     var elapsed2 = sw2.Elapsed;
832
833     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
    ↳ {elapsed2}");
834
835     Assert.True(elapsed1 > elapsed2);
836
837     // Checks
838     for (int i = START; i < END; i++)
839     {
840         var sequence1 = compressed1[i];
841         var sequence2 = compressed2[i];
842
843         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
844         {
845             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
    ↳ scope1.Links);
846
847             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
    ↳ scope2.Links);
848
849             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
850         }
851     }
852
853     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
854     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
855
856     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
    ↳ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
    ↳ totalCharacters}}");
857
858     // Can be worse than balanced variant
859     //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
860
861     //compressor1.ValidateFrequencies();
862 }

```

```

863 }
864
865 [Fact]
866 public static void AllTreeBreakDownAtSequencesCreationBugTest()
867 {
868     // Made out of AllPossibleConnectionsTest test.
869
870     //const long sequenceLength = 5; //100% bug
871     const long sequenceLength = 4; //100% bug
872     //const long sequenceLength = 3; //100% _no_bug (ok)
873
874     using (var scope = new TempLinksTestScope(useSequences: true))
875     {
876         var links = scope.Links;
877         var sequences = scope.Sequences;
878
879         var sequence = new ulong[sequenceLength];
880         for (var i = 0; i < sequenceLength; i++)
881         {
882             sequence[i] = links.Create();
883         }
884
885         var createResults = sequences.CreateAllVariants2(sequence);
886
887         Global.Trash = createResults;
888
889         for (var i = 0; i < sequenceLength; i++)
890         {
891             links.Delete(sequence[i]);
892         }
893     }
894 }
895
896 [Fact]
897 public static void AllPossibleConnectionsTest()
898 {
899     const long sequenceLength = 5;
900
901     using (var scope = new TempLinksTestScope(useSequences: true))
902     {
903         var links = scope.Links;
904         var sequences = scope.Sequences;
905
906         var sequence = new ulong[sequenceLength];
907         for (var i = 0; i < sequenceLength; i++)
908         {
909             sequence[i] = links.Create();
910         }
911
912         var createResults = sequences.CreateAllVariants2(sequence);
913         var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
914
915         for (var i = 0; i < 1; i++)
916         {
917             var sw1 = Stopwatch.StartNew();
918             var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
919
920             var sw2 = Stopwatch.StartNew();
921             var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
922
923             var sw3 = Stopwatch.StartNew();
924             var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
925
926             var sw4 = Stopwatch.StartNew();
927             var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
928
929             Global.Trash = searchResults3;
930             Global.Trash = searchResults4; //-V3008
931
932             var intersection1 = createResults.Intersect(searchResults1).ToList();
933             Assert.True(intersection1.Count == createResults.Length);
934
935             var intersection2 = reverseResults.Intersect(searchResults1).ToList();
936             Assert.True(intersection2.Count == reverseResults.Length);
937
938             var intersection0 = searchResults1.Intersect(searchResults2).ToList();
939             Assert.True(intersection0.Count == searchResults2.Count);
940
941             var intersection3 = searchResults2.Intersect(searchResults3).ToList();
942             Assert.True(intersection3.Count == searchResults3.Count);

```

```

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

```

#### 1.145 ./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
            ↳ MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
            ↳ implementation of tree cuts out 5 bits from the address space.
32         Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
            ↳ stMultipleRandomCreationsAndDeletions(100));
33         Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
            ↳ 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 }

```

#### 1.146 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt32LinksTests.cs

```

1  using System;
2  using Xunit;
3  using Platform.Memory;
4  using Platform.Data.Doublets.Memory.Split.Specific;
5  using TLink = System.UInt32;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public unsafe static class SplitMemoryUInt32LinksTests
10     {
11         [Fact]
12         public static void CRUDTest()
13         {
14             Using(links => links.TestCRUDOperations());
15         }
16
17         [Fact]
18         public static void RawNumbersCRUDTest()
19         {
20             UsingWithExternalReferences(links => links.TestRawNumbersCRUDOperations());
21         }
22
23         [Fact]
24         public static void MultipleRandomCreationsAndDeletionsTest()
25         {
26             Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultip
            ↳ leRandomCreationsAndDeletions(100));
27         }
28
29         private static void Using(Action<ILinks<TLink>> action)
30         {
31             using (var dataMemory = new HeapResizableDirectMemory())
32             using (var indexMemory = new HeapResizableDirectMemory())
33             using (var memory = new UInt32SplitMemoryLinks(dataMemory, indexMemory))
34             {

```

```

35         action(memory);
36     }
37 }
38
39 private static void UsingWithExternalReferences(Action<ILinks<TLink>> action)
40 {
41     var constants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
42     using (var dataMemory = new HeapResizableDirectMemory())
43     using (var indexMemory = new HeapResizableDirectMemory())
44     using (var memory = new UInt32SplitMemoryLinks(dataMemory, indexMemory,
45         ↪ UInt32SplitMemoryLinks.DefaultLinksSizeStep, constants))
46     {
47         action(memory);
48     }
49 }
50 }

```

#### 1.147 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt64LinksTests.cs

```

1  using System;
2  using Xunit;
3  using Platform.Memory;
4  using Platform.Data.Doublets.Memory.Split.Specific;
5  using TLink = System.UInt64;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public unsafe static class SplitMemoryUInt64LinksTests
10     {
11         [Fact]
12         public static void CRUDTest()
13         {
14             Using(links => links.TestCRUDOperations());
15         }
16
17         [Fact]
18         public static void RawNumbersCRUDTest()
19         {
20             UsingWithExternalReferences(links => links.TestRawNumbersCRUDOperations());
21         }
22
23         [Fact]
24         public static void MultipleRandomCreationsAndDeletionsTest()
25         {
26             Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(100));
27         }
28
29         private static void Using(Action<ILinks<TLink>> action)
30         {
31             using (var dataMemory = new HeapResizableDirectMemory())
32             using (var indexMemory = new HeapResizableDirectMemory())
33             using (var memory = new UInt64SplitMemoryLinks(dataMemory, indexMemory))
34             {
35                 action(memory);
36             }
37         }
38
39         private static void UsingWithExternalReferences(Action<ILinks<TLink>> action)
40         {
41             var constants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
42             using (var dataMemory = new HeapResizableDirectMemory())
43             using (var indexMemory = new HeapResizableDirectMemory())
44             using (var memory = new UInt64SplitMemoryLinks(dataMemory, indexMemory,
45                 ↪ UInt64SplitMemoryLinks.DefaultLinksSizeStep, constants))
46             {
47                 action(memory);
48             }
49         }
50     }

```

#### 1.148 ./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 }

```

#### 1.149 ./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     Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
28
29     var linkAddress = links.Create();
30
31     var link = new Link<T>(links.GetLink(linkAddress));
32
33     Assert.True(link.Count == 3);
34     Assert.True(equalityComparer.Equals(link.Index, linkAddress));
35     Assert.True(equalityComparer.Equals(link.Source, constants.Null));
36     Assert.True(equalityComparer.Equals(link.Target, constants.Null));
37
38     Assert.True(equalityComparer.Equals(links.Count(), one));
39
40     // Get first link
41     setter = new Setter<T>(constants.Null);
42     links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
43
44     Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
45
46     // Update link to reference itself
47     links.Update(linkAddress, linkAddress, linkAddress);
48
49     link = new Link<T>(links.GetLink(linkAddress));
50
51     Assert.True(equalityComparer.Equals(link.Source, linkAddress));
52     Assert.True(equalityComparer.Equals(link.Target, linkAddress));
53
54     // Update link to reference null (prepare for delete)
55     var updated = links.Update(linkAddress, constants.Null, constants.Null);
56
57     Assert.True(equalityComparer.Equals(updated, linkAddress));
58
59     link = new Link<T>(links.GetLink(linkAddress));
60
61     Assert.True(equalityComparer.Equals(link.Source, constants.Null));
62     Assert.True(equalityComparer.Equals(link.Target, constants.Null));
63
64     // Delete link
65     links.Delete(linkAddress);
66
67     Assert.True(equalityComparer.Equals(links.Count(), zero));
68
69     setter = new Setter<T>(constants.Null);
70     links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
71
72     Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
73 }
74
75
76 public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
77 {
78     // Constants
79     var constants = links.Constants;
80     var equalityComparer = EqualityComparer<T>.Default;
81
82     var zero = default(T);
83     var one = Arithmetic.Increment(zero);
84     var two = Arithmetic.Increment(one);
85
86     var h106E = new Hybrid<T>(106L, isExternal: true);
87     var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
88     var h108E = new Hybrid<T>(-108L);
89
90     Assert.Equal(106L, h106E.AbsoluteValue);
91     Assert.Equal(107L, h107E.AbsoluteValue);
92     Assert.Equal(108L, h108E.AbsoluteValue);
93
94     // Create Link (External -> External)
95     var linkAddress1 = links.Create();
96
97     links.Update(linkAddress1, h106E, h108E);
98
99     var link1 = new Link<T>(links.GetLink(linkAddress1));
100
101     Assert.True(equalityComparer.Equals(link1.Source, h106E));
102     Assert.True(equalityComparer.Equals(link1.Target, h108E));
103
104     // Create Link (Internal -> External)
105     var linkAddress2 = links.Create();
106

```

```

107     links.Update(linkAddress2, linkAddress1, h108E);
108
109     var link2 = new Link<T>(links.GetLink(linkAddress2));
110
111     Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
112     Assert.True(equalityComparer.Equals(link2.Target, h108E));
113
114     // Create Link (Internal -> Internal)
115     var linkAddress3 = links.Create();
116
117     links.Update(linkAddress3, linkAddress1, linkAddress2);
118
119     var link3 = new Link<T>(links.GetLink(linkAddress3));
120
121     Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
122     Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
123
124     // Search for created link
125     var setter1 = new Setter<T>(constants.Null);
126     links.Each(h106E, h108E, setter1.SetAndReturnFalse);
127
128     Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
129
130     // Search for nonexistent link
131     var setter2 = new Setter<T>(constants.Null);
132     links.Each(h106E, h107E, setter2.SetAndReturnFalse);
133
134     Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
135
136     // Update link to reference null (prepare for delete)
137     var updated = links.Update(linkAddress3, constants.Null, constants.Null);
138
139     Assert.True(equalityComparer.Equals(updated, linkAddress3));
140
141     link3 = new Link<T>(links.GetLink(linkAddress3));
142
143     Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
144     Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
145
146     // Delete link
147     links.Delete(linkAddress3);
148
149     Assert.True(equalityComparer.Equals(links.Count(), two));
150
151     var setter3 = new Setter<T>(constants.Null);
152     links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
153
154     Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
155 }
156
157 public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
    ↪ links, int maximumOperationsPerCycle)
158 {
159     var comparer = Comparer<TLink>.Default;
160     var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
161     var uint64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
162     for (var N = 1; N < maximumOperationsPerCycle; N++)
163     {
164         var random = new System.Random(N);
165         var created = 0UL;
166         var deleted = 0UL;
167         for (var i = 0; i < N; i++)
168         {
169             var linksCount = addressToUInt64Converter.Convert(links.Count());
170             var createPoint = random.NextBoolean();
171             if (linksCount > 2 && createPoint)
172             {
173                 var linksAddressRange = new Range<ulong>(1, linksCount);
174                 TLink source = uint64ToAddressConverter.Convert(random.NextUInt64(linksA
    ↪ ddressRange));
175                 TLink target = uint64ToAddressConverter.Convert(random.NextUInt64(linksA
    ↪ ddressRange));
176                 ↪ // -V3086
177                 var resultLink = links.GetOrCreate(source, target);
178                 if (comparer.Compare(resultLink,
    ↪ uint64ToAddressConverter.Convert(linksCount)) > 0)
179                 {
180                     created++;
181                 }
182             }
183         }
184     }
185 }

```

```

181         }
182         else
183         {
184             links.Create();
185             created++;
186         }
187     }
188     Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
189     for (var i = 0; i < N; i++)
190     {
191         TLink link = uInt64ToAddressConverter.Convert((ulong)i + 1UL);
192         if (links.Exists(link))
193         {
194             links.Delete(link);
195             deleted++;
196         }
197     }
198     Assert.True(addressToUInt64Converter.Convert(links.Count()) == 0L);
199 }
200 }
201 }
202 }

```

### 1.150 ./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
32         #region Concept
33
34         [Fact]
35         public static void MultipleCreateAndDeleteTest()
36         {
37             using (var scope = new Scope<Types<HeapResizableDirectMemory,
38                 ↪ UInt64UnitedMemoryLinks>>())
39             {
40                 new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeletions(100);
41             }
42         }
43
44         [Fact]
45         public static void CascadeUpdateTest()
46         {
47             var itself = _constants.Itself;
48             using (var scope = new TempLinksTestScope(useLog: true))
49             {
50                 var links = scope.Links;
51
52                 var l1 = links.Create();
53                 var l2 = links.Create();
54
55                 l2 = links.Update(l2, l2, l1, l2);
56             }
57         }
58     }
59 }

```

```

55         links.CreateAndUpdate(l2, itself);
56         links.CreateAndUpdate(l2, itself);
57
58         l2 = links.Update(l2, l1);
59
60         links.Delete(l2);
61
62         Global.Trash = links.Count();
63
64         links.Unsync.DisposeIfPossible(); // Close links to access log
65
66         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
        ↪ e.TempTransactionLogFilename);
67     }
68 }
69
70 [Fact]
71 public static void BasicTransactionLogTest()
72 {
73     using (var scope = new TempLinksTestScope(useLog: true))
74     {
75         var links = scope.Links;
76         var l1 = links.Create();
77         var l2 = links.Create();
78
79         Global.Trash = links.Update(l2, l2, l1, l2);
80
81         links.Delete(l1);
82
83         links.Unsync.DisposeIfPossible(); // Close links to access log
84
85         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
        ↪ e.TempTransactionLogFilename);
86     }
87 }
88
89 [Fact]
90 public static void TransactionAutoRevertedTest()
91 {
92     // Auto Reverted (Because no commit at transaction)
93     using (var scope = new TempLinksTestScope(useLog: true))
94     {
95         var links = scope.Links;
96         var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
97         using (var transaction = transactionsLayer.BeginTransaction())
98         {
99             var l1 = links.Create();
100             var l2 = links.Create();
101
102             links.Update(l2, l2, l1, l2);
103         }
104
105         Assert.Equal(OUL, links.Count());
106
107         links.Unsync.DisposeIfPossible();
108
109         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
        ↪ cope.TempTransactionLogFilename);
110         Assert.Single(transitions);
111     }
112 }
113
114 [Fact]
115 public static void TransactionUserCodeErrorNoDataSavedTest()
116 {
117     // User Code Error (Autoreverted), no data saved
118     var itself = _constants.Itself;
119
120     TempLinksTestScope lastScope = null;
121     try
122     {
123         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
        ↪ useLog: true))
124         {
125             var links = scope.Links;
126             var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
        ↪ atorBase<ulong>)links.Unsync).Links;
127             using (var transaction = transactionsLayer.BeginTransaction())
128             {

```

```

129         var l1 = links.CreateAndUpdate(itself, itself);
130         var l2 = links.CreateAndUpdate(itself, itself);
131
132         l2 = links.Update(l2, l2, l1, l2);
133
134         links.CreateAndUpdate(l2, itself);
135         links.CreateAndUpdate(l2, itself);
136
137         //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
138         ↪ tion>(scope.TempTransactionLogFilename);
139
140         l2 = links.Update(l2, l1);
141
142         links.Delete(l2);
143
144         ExceptionThrower();
145
146         transaction.Commit();
147     }
148     Global.Trash = links.Count();
149 }
150 }
151 catch
152 {
153     Assert.False(lastScope == null);
154
155     var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
156     ↪ astScope.TempTransactionLogFilename);
157
158     Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
159     ↪ transitions[0].After.IsNull());
160
161     lastScope.DeleteFiles();
162 }
163 }
164 [Fact]
165 public static void TransactionUserCodeErrorSomeDataSavedTest()
166 {
167     // User Code Error (Autoreverted), some data saved
168     var itself = _constants.Itself;
169
170     TempLinksTestScope lastScope = null;
171     try
172     {
173         ulong l1;
174         ulong l2;
175
176         using (var scope = new TempLinksTestScope(useLog: true))
177         {
178             var links = scope.Links;
179             l1 = links.CreateAndUpdate(itself, itself);
180             l2 = links.CreateAndUpdate(itself, itself);
181
182             l2 = links.Update(l2, l2, l1, l2);
183
184             links.CreateAndUpdate(l2, itself);
185             links.CreateAndUpdate(l2, itself);
186
187             links.Unsync.DisposeIfPossible();
188
189             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
190             ↪ scope.TempTransactionLogFilename);
191         }
192
193         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
194         ↪ useLog: true))
195         {
196             var links = scope.Links;
197             var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
198             using (var transaction = transactionsLayer.BeginTransaction())
199             {
200                 l2 = links.Update(l2, l1);
201
202                 links.Delete(l2);
203
204                 ExceptionThrower();
205
206                 transaction.Commit();

```



```

204         }
205     }
206     Global.Trash = links.Count();
207 }
208 }
209 catch
210 {
211     Assert.False(lastScope == null);
212     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
        ↪ Scope.TempTransactionLogFilename);
213     lastScope.DeleteFiles();
214 }
215 }
216 }
217 }
218
219 [Fact]
220 public static void TransactionCommit()
221 {
222     var itself = _constants.Itself;
223
224     var tempDatabaseFilename = Path.GetTempFileName();
225     var tempTransactionLogFilename = Path.GetTempFileName();
226
227     // Commit
228     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
        ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
229     using (var links = new UInt64Links(memoryAdapter))
230     {
231         using (var transaction = memoryAdapter.BeginTransaction())
232         {
233             var l1 = links.CreateAndUpdate(itself, itself);
234             var l2 = links.CreateAndUpdate(itself, itself);
235
236             Global.Trash = links.Update(l2, l2, l1, l2);
237
238             links.Delete(l1);
239
240             transaction.Commit();
241         }
242
243         Global.Trash = links.Count();
244     }
245
246     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
        ↪ sactionLogFilename);
247 }
248
249 [Fact]
250 public static void TransactionDamage()
251 {
252     var itself = _constants.Itself;
253
254     var tempDatabaseFilename = Path.GetTempFileName();
255     var tempTransactionLogFilename = Path.GetTempFileName();
256
257     // Commit
258     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
        ↪ 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,
            ↳ tempTransactionLogFilename))
319         using (var links = new UInt64Links(memoryAdapter))
320         {
321             l1 = links.CreateAndUpdate(itself, itself);
322             l2 = links.CreateAndUpdate(itself, itself);
323
324             l2 = links.Update(l2, l2, l1, l2);
325
326             links.CreateAndUpdate(l2, itself);
327             links.CreateAndUpdate(l2, itself);
328         }
329
330         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp_
            ↳ TransactionLogFilename);
331
332         using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
333         using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
            ↳ tempTransactionLogFilename))
334         using (var links = new UInt64Links(memoryAdapter))
335         {
336             using (var transaction = memoryAdapter.BeginTransaction())
337             {
338                 l2 = links.Update(l2, l1);
339
340                 links.Delete(l2);
341
342                 ExceptionThrower();
343
344                 transaction.Commit();
345             }
346
347             Global.Trash = links.Count();
348         }
349     }

```

```

350     catch
351     {
352         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp ↵
            ↵ TransactionLogFilename);
353     }
354
355     File.Delete(tempDatabaseFilename);
356     File.Delete(tempTransactionLogFilename);
357 }
358
359 private static void ExceptionThrower() => throw new InvalidOperationException();
360
361 [Fact]
362 public static void PathsTest()
363 {
364     var source = _constants.SourcePart;
365     var target = _constants.TargetPart;
366
367     using (var scope = new TempLinksTestScope())
368     {
369         var links = scope.Links;
370         var l1 = links.CreatePoint();
371         var l2 = links.CreatePoint();
372
373         var r1 = links.GetByKeys(l1, source, target, source);
374         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
375     }
376 }
377
378 [Fact]
379 public static void RecursiveStringFormattingTest()
380 {
381     using (var scope = new TempLinksTestScope(useSequences: true))
382     {
383         var links = scope.Links;
384         var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
385
386         var a = links.CreatePoint();
387         var b = links.CreatePoint();
388         var c = links.CreatePoint();
389
390         var ab = links.GetOrCreate(a, b);
391         var cb = links.GetOrCreate(c, b);
392         var ac = links.GetOrCreate(a, c);
393
394         a = links.Update(a, c, b);
395         b = links.Update(b, a, c);
396         c = links.Update(c, a, b);
397
398         Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
399         Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
400         Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
401
402         Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
            ↵ "(5:(4:5 (6:5 4)) 6)");
403         Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
            ↵ "(6:(5:(4:5 6) 6) 4)");
404         Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
            ↵ "(4:(5:4 (6:5 4)) 6)");
405
406         // TODO: Think how to build balanced syntax tree while formatting structure (eg.
            ↵ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)"
407
408         Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
            ↵ "{{5}{5}{4}{6}}");
409         Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
            ↵ "{{5}{6}{6}{4}}");
410         Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
            ↵ "{{4}{5}{4}{6}}");
411     }
412 }
413
414 private static void DefaultFormatter(StringBuilder sb, ulong link)
415 {
416     sb.Append(link.ToString());
417 }
418
419 #endregion
420

```

## #region Performance

```
/*
public static void RunAllPerformanceTests()
{
    try
    {
        links.TestLinksInSteps();
    }
    catch (Exception ex)
    {
        ex.WriteToConsole();
    }

    return;

    try
    {
        //ThreadPool.SetMaxThreads(2, 2);

        // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
        // Также это дополнительно помогает в отладке
        // Увеличивает вероятность попадания информации в кэши
        for (var i = 0; i < 10; i++)
        {
            //0 - 10 ГБ
            //Каждые 100 МБ срез цифр

            //links.TestGetSourceFunction();
            //links.TestGetSourceFunctionInParallel();
            //links.TestGetTargetFunction();
            //links.TestGetTargetFunctionInParallel();
            links.Create64BillionLinks();

            links.TestRandomSearchFixed();
            //links.Create64BillionLinksInParallel();
            links.TestEachFunction();
            //links.TestForeach();
            //links.TestParallelForeach();
        }

        links.TestDeletionOfAllLinks();

    }
    catch (Exception ex)
    {
        ex.WriteToConsole();
    }
}*/

/*
public static void TestLinksInSteps()
{
    const long gibibyte = 1024 * 1024 * 1024;
    const long mebibyte = 1024 * 1024;

    var totalLinksToCreate = gibibyte /
    Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
    var linksStep = 102 * mebibyte /
    Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;

    var creationMeasurements = new List<TimeSpan>();
    var searchMeasurements = new List<TimeSpan>();
    var deletionMeasurements = new List<TimeSpan>();

    GetBaseRandomLoopOverhead(linksStep);
    GetBaseRandomLoopOverhead(linksStep);

    var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);

    ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);

    var loops = totalLinksToCreate / linksStep;

    for (int i = 0; i < loops; i++)
    {
        creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
        searchMeasurements.Add(Measure(() => links.RunRandomSearches(linksStep)));
    }
}
```

```

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

```

```

572         counter += links.GetSource(firstLink);
573     }
574
575     var elapsedTime = sw.Elapsed;
576
577     var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
578
579     // Удаляем связь, из которой производилось считывание
580     links.Delete(firstLink);
581
582     ConsoleHelpers.Debug(
583         "{0} Iterations of GetSource function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
584     }
585 }
586
587 [Fact(Skip = "performance test")]
588 public static void GetSourceInParallel()
589 {
590     using (var scope = new TempLinksTestScope())
591     {
592         var links = scope.Links;
593         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
        ↳ parallel.", Iterations);
594
595         long counter = 0;
596
597         //var firstLink = links.First();
598         var firstLink = links.Create();
599
600         var sw = Stopwatch.StartNew();
601
602         // Тестируем саму функцию
603         Parallel.For(0, Iterations, x =>
604         {
605             Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
606             //Interlocked.Increment(ref counter);
607         });
608
609         var elapsedTime = sw.Elapsed;
610
611         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
612
613         links.Delete(firstLink);
614
615         ConsoleHelpers.Debug(
616             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
617     }
618 }
619
620 [Fact(Skip = "performance test")]
621 public static void TestGetTarget()
622 {
623     using (var scope = new TempLinksTestScope())
624     {
625         var links = scope.Links;
626         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
        ↳ Iterations);
627
628         ulong counter = 0;
629
630         //var firstLink = links.First();
631         var firstLink = links.Create();
632
633         var sw = Stopwatch.StartNew();
634
635         for (ulong i = 0; i < Iterations; i++)
636         {
637             counter += links.GetTarget(firstLink);
638         }
639
640         var elapsedTime = sw.Elapsed;
641
642         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
643
644         links.Delete(firstLink);
645     }
646 }
647

```

```

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

```

```

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.",
736             ↪ links.Count());
737
738         var sw = Stopwatch.StartNew();
739
740         for (var i = iterations; i > 0; i--)
741         {
742             var linksAddressRange = new
743                 ↪ Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);
744
745             var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
746             var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
747
748             counter += links.SearchOrDefault(source, target);
749         }
750
751         var elapsedTime = sw.Elapsed;
752
753         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
754
755         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
756             ↪ Iterations per second), c: {3}",
757             iterations, elapsedTime, (long)iterationsPerSecond, counter);
758     }
759 }
760
761 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
762 public static void TestEach()
763 {
764     using (var scope = new TempLinksTestScope())
765     {
766         var links = scope.Links;
767
768         var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
769
770         ConsoleHelpers.Debug("Testing Each function.");
771
772         var sw = Stopwatch.StartNew();
773
774         links.Each(counter.IncrementAndReturnTrue);
775
776         var elapsedTime = sw.Elapsed;
777
778         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
779
780         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
781             ↪ links per second)",
782             counter, elapsedTime, (long)linksPerSecond);
783     }
784 }
785
786 /*
787 [Fact]
788 public static void TestForeach()
789 {
790     var tempFilename = Path.GetTempFileName();
791
792     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
793         ↪ DefaultLinksSizeStep))
794     {
795         ulong counter = 0;
796
797         ConsoleHelpers.Debug("Testing foreach through links.");
798     }
799 }

```



```

793         var sw = Stopwatch.StartNew();
794
795         //foreach (var link in links)
796         //{
797             //    counter++;
798         //}
799
800         var elapsedTime = sw.Elapsed;
801
802         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
803
804         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
↪ links per second)", counter, elapsedTime, (long)linksPerSecond);
805     }
806
807     File.Delete(tempFilename);
808 }
809 */
810
811 /*
812 [Fact]
813 public static void TestParallelForeach()
814 {
815     var tempFilename = Path.GetTempFileName();
816
817     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪ DefaultLinksSizeStep))
818     {
819         long counter = 0;
820
821         ConsoleHelpers.Debug("Testing parallel foreach through links.");
822
823         var sw = Stopwatch.StartNew();
824
825         //Parallel.ForEach((IEnumerable<ulong>)links, x =>
826         //{
827             //    Interlocked.Increment(ref counter);
828         //});
829
830         var elapsedTime = sw.Elapsed;
831
832         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
833
834         ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↪ {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
835     }
836
837     File.Delete(tempFilename);
838 }
839 */
840
841 [Fact(Skip = "performance test")]
842 public static void Create64BillionLinks()
843 {
844     using (var scope = new TempLinksTestScope())
845     {
846         var links = scope.Links;
847         var linksBeforeTest = links.Count();
848
849         long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;
850
851         ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
852
853         var elapsedTime = Performance.Measure(() =>
854         {
855             for (long i = 0; i < linksToCreate; i++)
856             {
857                 links.Create();
858             }
859         });
860
861         var linksCreated = links.Count() - linksBeforeTest;
862         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
863
864         ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
865
866         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
↪ linksCreated, elapsedTime,

```

```

869         (long)linksPerSecond);
870     }
871 }
872
873 [Fact(Skip = "performance test")]
874 public static void Create64BillionLinksInParallel()
875 {
876     using (var scope = new TempLinksTestScope())
877     {
878         var links = scope.Links;
879         var linksBeforeTest = links.Count();
880
881         var sw = Stopwatch.StartNew();
882
883         long linksToCreate = 64 * 1024 * 1024 / 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 }

```

#### 1.151 ./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];

```

```

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

```

### 1.152 ./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 using Platform.Data.Doublets.CriterionMatchers;
16
17 namespace Platform.Data.Doublets.Tests
18 {
19     public static class UnicodeConvertersTests
20     {
21         [Fact]
22         public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
23         {
24             using (var scope = new TempLinksTestScope())
25             {
26                 var links = scope.Links;
27                 var meaningRoot = links.CreatePoint();
28                 var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
29                 var powerOf2ToUnaryNumberConverter = new
30                 ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
31                 var addressToUnaryNumberConverter = new
32                 ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
33                 var unaryNumberToAddressConverter = new
34                 ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
35                 ↪ powerOf2ToUnaryNumberConverter);
36                 TestCharAndUnicodeSymbolConverters(links, meaningRoot,
37                 ↪ addressToUnaryNumberConverter, unaryNumberToAddressConverter);
38             }
39         }
40
41         [Fact]
42         public static void CharAndRawNumberUnicodeSymbolConvertersTest()
43         {
44             using (var scope = new Scope<Types<HeapResizableDirectMemory,
45             ↪ UnitedMemoryLinks<ulong>>>())
46             {
47                 var links = scope.Use<ILinks<ulong>>>();
48                 var meaningRoot = links.CreatePoint();
49                 var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
50                 var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
51                 TestCharAndUnicodeSymbolConverters(links, meaningRoot,
52                 ↪ addressToRawNumberConverter, rawNumberToAddressConverter);
53             }
54         }
55     }
56 }

```

```

49 private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
    ↳ meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
    ↳ numberToAddressConverter)
50 {
51     var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
52     var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
    ↳ addressToNumberConverter, unicodeSymbolMarker);
53     var originalCharacter = 'H';
54     var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
55     var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
    ↳ unicodeSymbolMarker);
56     var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
    ↳ numberToAddressConverter, unicodeSymbolCriterionMatcher);
57     var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
58     Assert.Equal(originalCharacter, resultingCharacter);
59 }
60
61 [Fact]
62 public static void StringAndUnicodeSequenceConvertersTest()
63 {
64     using (var scope = new TempLinksTestScope())
65     {
66         var links = scope.Links;
67
68         var itself = links.Constants.Itself;
69
70         var meaningRoot = links.CreatePoint();
71         var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
72         var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
73         var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
74         var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
75         var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
76
77         var powerOf2ToUnaryNumberConverter = new
    ↳ PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
78         var addressToUnaryNumberConverter = new
    ↳ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
79         var charToUnicodeSymbolConverter = new
    ↳ CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
    ↳ unicodeSymbolMarker);
80
81         var unaryNumberToAddressConverter = new
    ↳ UnaryNumberToAddressOrOperationConverter<ulong>(links,
    ↳ powerOf2ToUnaryNumberConverter);
82         var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
83         var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
    ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
84         var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
    ↳ frequencyPropertyMarker, frequencyMarker);
85         var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
    ↳ frequencyPropertyOperator, frequencyIncrementer);
86         var linkToItsFrequencyNumberConverter = new
    ↳ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
    ↳ unaryNumberToAddressConverter);
87         var sequenceToItsLocalElementLevelsConverter = new
    ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
    ↳ linkToItsFrequencyNumberConverter);
88         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
    ↳ sequenceToItsLocalElementLevelsConverter);
89
90         var stringToUnicodeSequenceConverter = new
    ↳ StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
    ↳ index, optimalVariantConverter, unicodeSequenceMarker);
91
92         var originalString = "Hello";
93
94         var unicodeSequenceLink =
    ↳ stringToUnicodeSequenceConverter.Convert(originalString);
95
96         var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
    ↳ unicodeSymbolMarker);
97         var unicodeSymbolToCharConverter = new
    ↳ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
    ↳ unicodeSymbolCriterionMatcher);
98
99         var unicodeSequenceCriterionMatcher = new TargetMatcher<ulong>(links,
    ↳ unicodeSequenceMarker);
100

```

```

101         var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
102             ↳ unicodeSymbolCriterionMatcher.IsMatched);
103         var unicodeSequenceToStringConverter = new
104             ↳ UnicodeSequenceToStringConverter<ulong>(links,
105             ↳ unicodeSequenceCriterionMatcher, sequenceWalker,
106             ↳ unicodeSymbolToCharConverter);
107         var resultingString =
108             ↳ unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
109         Assert.Equal(originalString, resultingString);
110     }
111 }

```

#### 1.153 ./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt32LinksTests.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.Specific;
7  using TLink = System.UInt32;
8
9  namespace Platform.Data.Doublets.Tests
10 {
11     public unsafe static class UnitedMemoryUInt32LinksTests
12     {
13         [Fact]
14         public static void CRUDTest()
15         {
16             Using(links => links.TestCRUDOperations());
17         }
18
19         [Fact]
20         public static void RawNumbersCRUDTest()
21         {
22             Using(links => links.TestRawNumbersCRUDOperations());
23         }
24
25         [Fact]
26         public static void MultipleRandomCreationsAndDeletionsTest()
27         {
28             Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(100));
29         }
30
31         private static void Using(Action<ILinks<TLink>> action)
32         {
33             using (var scope = new Scope<Types<HeapResizableDirectMemory,
34                 ↳ UInt32UnitedMemoryLinks>>())
35             {
36                 action(scope.Use<ILinks<TLink>>());
37             }
38         }
39     }

```

#### 1.154 ./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt64LinksTests.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.Specific;
7  using TLink = System.UInt64;
8
9  namespace Platform.Data.Doublets.Tests
10 {
11     public unsafe static class UnitedMemoryUInt64LinksTests
12     {
13         [Fact]
14         public static void CRUDTest()
15         {
16             Using(links => links.TestCRUDOperations());
17         }
18     }

```

```

19     [Fact]
20     public static void RawNumbersCRUDTest()
21     {
22         Using(links => links.TestRawNumbersCRUDOperations());
23     }
24
25     [Fact]
26     public static void MultipleRandomCreationsAndDeletionsTest()
27     {
28         Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(100));
29     }
30
31     private static void Using(Action<ILinks<TLink>> action)
32     {
33         using (var scope = new Scope<Types<HeapResizableDirectMemory,
34             ↳ UInt64UnitedMemoryLinks>>())
35         {
36             action(scope.Use<ILinks<TLink>>());
37         }
38     }
39 }

```

## Index

`./csharp/Platform.Data.Doublets.Tests/GenericLinksTests.cs`, 195  
`./csharp/Platform.Data.Doublets.Tests/ILinksExtensionsTests.cs`, 195  
`./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs`, 196  
`./csharp/Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs`, 196  
`./csharp/Platform.Data.Doublets.Tests/ReadSequenceTests.cs`, 199  
`./csharp/Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs`, 200  
`./csharp/Platform.Data.Doublets.Tests/ScopeTests.cs`, 201  
`./csharp/Platform.Data.Doublets.Tests/SequencesTests.cs`, 202  
`./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs`, 216  
`./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt32LinksTests.cs`, 217  
`./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt64LinksTests.cs`, 218  
`./csharp/Platform.Data.Doublets.Tests/TempLinksTestScope.cs`, 218  
`./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs`, 219  
`./csharp/Platform.Data.Doublets.Tests/UInt64LinksTests.cs`, 222  
`./csharp/Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs`, 234  
`./csharp/Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs`, 235  
`./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt32LinksTests.cs`, 237  
`./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt64LinksTests.cs`, 237  
`./csharp/Platform.Data.Doublets/CriterionMatchers/TargetMatcher.cs`, 1  
`./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`, 5  
`./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/UInt32Links.cs`, 6  
`./csharp/Platform.Data.Doublets/Decorators/UInt64Links.cs`, 7  
`./csharp/Platform.Data.Doublets/Decorators/UniLinks.cs`, 8  
`./csharp/Platform.Data.Doublets/Doublet.cs`, 13  
`./csharp/Platform.Data.Doublets/DoubletComparer.cs`, 13  
`./csharp/Platform.Data.Doublets/ILinks.cs`, 14  
`./csharp/Platform.Data.Doublets/ILinksExtensions.cs`, 14  
`./csharp/Platform.Data.Doublets/ISynchronizedLinks.cs`, 26  
`./csharp/Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs`, 26  
`./csharp/Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs`, 27  
`./csharp/Platform.Data.Doublets/Link.cs`, 27  
`./csharp/Platform.Data.Doublets/LinkExtensions.cs`, 30  
`./csharp/Platform.Data.Doublets/LinksOperatorBase.cs`, 31  
`./csharp/Platform.Data.Doublets/Memory/ILinksListMethods.cs`, 31  
`./csharp/Platform.Data.Doublets/Memory/ILinksTreeMethods.cs`, 31  
`./csharp/Platform.Data.Doublets/Memory/IndexTreeType.cs`, 31  
`./csharp/Platform.Data.Doublets/Memory/LinksHeader.cs`, 32  
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSizeBalancedTreeMethodsBase.cs`, 32  
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSourcesSizeBalancedTreeMethods.cs`, 36  
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsSizeBalancedTreeMethods.cs`, 37  
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSizeBalancedTreeMethodsBase.cs`, 37  
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesSizeBalancedTreeMethods.cs`, 40  
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsSizeBalancedTreeMethods.cs`, 41  
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinks.cs`, 42  
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinksBase.cs`, 43  
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/UnusedLinksListMethods.cs`, 53  
`./csharp/Platform.Data.Doublets/Memory/Split/RawLinkDataPart.cs`, 54  
`./csharp/Platform.Data.Doublets/Memory/Split/RawLinkIndexPart.cs`, 54  
`./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSizeBalancedTreeMethodsBase.cs`, 55  
`./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSourcesSizeBalancedTreeMethods.cs`, 56  
`./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksTargetsSizeBalancedTreeMethods.cs`, 57  
`./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSizeBalancedTreeMethodsBase.cs`, 58  
`./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesSizeBalancedTreeMethods.cs`, 59  
`./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksTargetsSizeBalancedTreeMethods.cs`, 60  
`./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32SplitMemoryLinks.cs`, 61

./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32UnusedLinksListMethods.cs, 63  
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSizeBalancedTreeMethodsBase.cs, 63  
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSourcesSizeBalancedTreeMethods.cs, 65  
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksTargetsSizeBalancedTreeMethods.cs, 66  
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSizeBalancedTreeMethodsBase.cs, 66  
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesSizeBalancedTreeMethods.cs, 68  
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksTargetsSizeBalancedTreeMethods.cs, 69  
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64SplitMemoryLinks.cs, 69  
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64UnusedLinksListMethods.cs, 71  
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksAvlBalancedTreeMethodsBase.cs, 71  
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSizeBalancedTreeMethodsBase.cs, 76  
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesAvlBalancedTreeMethods.cs, 79  
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesSizeBalancedTreeMethods.cs, 80  
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsAvlBalancedTreeMethods.cs, 81  
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsSizeBalancedTreeMethods.cs, 82  
./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinks.cs, 83  
./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinksBase.cs, 84  
./csharp/Platform.Data.Doublets/Memory/United/Generic/UnusedLinksListMethods.cs, 91  
./csharp/Platform.Data.Doublets/Memory/United/RawLink.cs, 92  
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSizeBalancedTreeMethodsBase.cs, 93  
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSourcesSizeBalancedTreeMethods.cs, 94  
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksTargetsSizeBalancedTreeMethods.cs, 95  
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32UnitedMemoryLinks.cs, 96  
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32UnusedLinksListMethods.cs, 97  
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs, 98  
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs, 99  
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs, 101  
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.cs, 102  
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs, 103  
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs, 104  
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnitedMemoryLinks.cs, 105  
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnusedLinksListMethods.cs, 107  
./csharp/Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs, 107  
./csharp/Platform.Data.Doublets/Numbers/Unary/LinkToToltsFrequencyNumberConveter.cs, 108  
./csharp/Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs, 108  
./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs, 109  
./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs, 110  
./csharp/Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 111  
./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 112  
./csharp/Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 113  
./csharp/Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 113  
./csharp/Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 117  
./csharp/Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 117  
./csharp/Platform.Data.Doublets/Sequences/Converters/SequenceToToltsLocalElementLevelsConverter.cs, 119  
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs, 119  
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs, 119  
./csharp/Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 120  
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 120  
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 121  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 123  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 125  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToToltsFrequencyValueConverter.cs, 126  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 126  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 126  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 127  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 127  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 128  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 128  
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 129  
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 130  
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 130  
./csharp/Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 130  
./csharp/Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 131  
./csharp/Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 132  
./csharp/Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 132  
./csharp/Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 133  
./csharp/Platform.Data.Doublets/Sequences/Indexes/Unindex.cs, 134



./csharp/Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 134  
./csharp/Platform.Data.Doublets/Sequences/Sequences.cs, 161  
./csharp/Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 172  
./csharp/Platform.Data.Doublets/Sequences/SequencesOptions.cs, 172  
./csharp/Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 175  
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 175  
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 176  
./csharp/Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 177  
./csharp/Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 178  
./csharp/Platform.Data.Doublets/Stacks/Stack.cs, 179  
./csharp/Platform.Data.Doublets/Stacks/StackExtensions.cs, 180  
./csharp/Platform.Data.Doublets/SynchronizedLinks.cs, 180  
./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs, 181  
./csharp/Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 183  
./csharp/Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 189  
./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 189  
./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSymbolsListConverter.cs, 190  
./csharp/Platform.Data.Doublets/Unicode/UnicodeMap.cs, 190  
./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 193  
./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 194  
./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.cs, 194