

## 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
18            {
19                get { return true; }
20            }
21        }
22    }
23 }
```

```

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(ICollection<TLink> restrictions) => _links.CreatePoint();
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public override TLink Update(ICollection<TLink> restrictions, ICollection<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(ICollection<TLink> links) : base(links) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public override TLink Update(ICollection<TLink> restrictions, ICollection<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(ICollection<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public override TLink Update(ICollection<TLink> restrictions, ICollection<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(ICollection<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 }
82
83 }

```

## 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     ///        }
90     ///    }
91     ///    if (!Memory.Each(handler, restriction))
92     ///    {
93     ///        return Constants.Break;
94     ///    }
95 }

```

```

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>
578     ↳ links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
579 {
580     var equalityComparer = EqualityComparer<TLink>.Default;
581     var constants = links.Constants;
582     var restrictionsIndex = restrictions[constants.IndexPart];
583     var substitutionIndex = substitution[constants.IndexPart];
584     if (equalityComparer.Equals(substitutionIndex, default))
585     {
586         substitutionIndex = restrictionsIndex;
587     }
588     var source = substitution[constants.SourcePart];
589     var target = substitution[constants.TargetPart];
590     source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
591     target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
592     return new Link<TLink>(substitutionIndex, source, target);
593 }
594
595 /// <summary>
596 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
597   ↳ с указанными Source (началом) и Target (концом).
598   ↳ </summary>
599   ↳ <param name="links">Хранилище связей.</param>
600   ↳ <param name="source">Индекс связи, которая является началом на создаваемой
601     ↳ связи.</param>
602   ↳ <param name="target">Индекс связи, которая является концом для создаваемой
603     ↳ связи.</param>
604   ↳ <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
605 [MethodImpl(MethodImplOptions.AggressiveInlining)]
606 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
607     ↳ target)
608 {
609     var link = links.SearchOrDefault(source, target);
610     if (EqualityComparer<TLink>.Default.Equals(link, default))
611     {
612         link = links.CreateAndUpdate(source, target);
613     }
614     return link;
615 }
616
617 /// <summary>
618 /// Обновляет связь с указанными началом (Source) и концом (Target)
619   ↳ на связь с указанными началом (NewSource) и концом (NewTarget).
620   ↳ </summary>
621   ↳ <param name="links">Хранилище связей.</param>
622   ↳ <param name="source">Индекс связи, которая является началом обновляемой
623     ↳ связи.</param>
624   ↳ <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
625   ↳ <param name="newSource">Индекс связи, которая является началом связи, на которую
626     ↳ выполняется обновление.</param>
627   ↳ <param name="newTarget">Индекс связи, которая является концом связи, на которую
628     ↳ выполняется обновление.</param>
629   ↳ <returns>Индекс обновлённой связи.</returns>
630 [MethodImpl(MethodImplOptions.AggressiveInlining)]
631 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
632     ↳ TLink target, TLink newSource, TLink newTarget)
633 {
634     var equalityComparer = EqualityComparer<TLink>.Default;
635     var link = links.SearchOrDefault(source, target);
636     if (equalityComparer.Equals(link, default))
637     {
638         return links.CreateAndUpdate(newSource, newTarget);
639     }
640     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
641     ↳ target))
642     {
643         return link;
644     }
645     return links.Update(link, newSource, newTarget);
646 }
647
648 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
649   ↳ <param name="links">Хранилище связей.</param>
650   ↳ <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
651   ↳ <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
652 [MethodImpl(MethodImplOptions.AggressiveInlining)]
653 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
654     ↳ 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                 ArrayPool.Free(usages);
788             }
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;
23     }
24 }

```

```

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/ExternalLinksRecursionlessSizeBalancedTreeMethod

```
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 ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink> :
14         ↳ RecursionlessSizeBalancedTreeMethods<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 ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
27             ↳ constants, 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/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         {

```

```

64     ref var firstLink = ref GetLinkDataPartReference(first);
65     ref var secondLink = ref GetLinkDataPartReference(second);
66     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
67 }
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
71 {
72     ref var firstLink = ref GetLinkDataPartReference(first);
73     ref var secondLink = ref GetLinkDataPartReference(second);
74     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
75 }
76
77 public TLink this[TLink index]
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
    ↪ broken)
104     }
105 }
106
107 /// <summary>
108 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
    ↪ (концом).
109 /// </summary>
110 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
111 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
112 /// <returns>Индекс искомой связи.</returns>
113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
114 public TLink Search(TLink source, TLink target)
115 {
116     var root = GetTreeRoot();
117     while (!EqualToZero(root))
118     {
119         ref var rootLink = ref GetLinkDataPartReference(root);
120         var rootSource = rootLink.Source;
121         var rootTarget = rootLink.Target;
122         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
    ↪ node.Key < root.Key
123         {
124             root = GetLeftOrDefault(root);
125         }
126         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
    ↪ node.Key > root.Key
127         {
128             root = GetRightOrDefault(root);
129         }
130         else // node.Key == root.Key
131         {
132             return root;
133         }
134     }
135     return Zero;

```

```

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

```

```

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

```

### 1.33 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSourcesRecursionlessSizeBalancedTree

```

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 ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLink> :
8         ↳ ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
12             ↳ constants, byte* linksDataParts, byte* linksIndexParts, byte* header) :
13             ↳ base(constants, 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)]

```

```

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

```

### 1.34 ./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 }

```



```

44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
46     ↪ 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 GetLinkIndexPartReference(node);
52         link.LeftAsSource = Zero;
53         link.RightAsSource = Zero;
54         link.SizeAsSource = Zero;
55     }
56 }
57 }

```

### 1.35 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsRecursionlessSizeBalancedTree

```

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 ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods<TLink> :
8     ↪ ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
12         ↪ constants, byte* linksDataParts, byte* linksIndexParts, byte* header) :
13         ↪ base(constants, 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 FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
56         ↪ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
57         ↪ (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

### 1.36 ./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 FirstIsToTheLeftOfSecond(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 }

```

```

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

```

### 1.37 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksRecursionlessSizeBalancedTreeMethod

```

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 InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink> :
14         ↳ RecursionlessSizeBalancedTreeMethods<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 InternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
27             ↳ constants, byte* linksDataParts, byte* linksIndexParts, byte* header)
28         {
29             LinksDataParts = linksDataParts;
30             LinksIndexParts = linksIndexParts;
31             Header = header;
32             Break = constants.Break;
33             Continue = constants.Continue;
34         }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected abstract TLink GetTreeRoot(TLink link);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected abstract TLink GetBasePartValue(TLink link);
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected abstract TLink GetKeyPartValue(TLink link);
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
47             ↳ AsRef<RawLinkDataPart<TLink>>(LinksDataParts + (RawLinkDataPart<TLink>.SizeInBytes *
48                 ↳ _addressToInt64Converter.Convert(link)));
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
52             ↳ ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
53                 ↳ (RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link)));
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override bool FirstIsToLeftOfSecond(TLink first, TLink second) =>
57             ↳ LessThan(GetKeyPartValue(first), GetKeyPartValue(second));
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
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 }

```

```

59     }
60
61     public TLink this[TLink link, TLink index]
62     {
63         [MethodImpl(MethodImplOptions.AggressiveInlining)]
64         get
65         {
66             var root = GetTreeRoot(link);
67             if (GreaterOrEqualThan(index, GetSize(root)))
68             {
69                 return Zero;
70             }
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
92     /// <summary>
93     /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
94     /// ↳ (концом).
95     /// </summary>
96     /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
97     /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
98     /// <returns>Индекс искомой связи.</returns>
99     [MethodImpl(MethodImplOptions.AggressiveInlining)]
100     public abstract TLink Search(TLink source, TLink target);
101
102     [MethodImpl(MethodImplOptions.AggressiveInlining)]
103     protected TLink SearchCore(TLink root, TLink key)
104     {
105         while (!EqualToZero(root))
106         {
107             var rootKey = GetKeyPartValue(root);
108             if (LessThan(key, rootKey)) // node.Key < root.Key
109             {
110                 root = GetLeftOrDefault(root);
111             }
112             else if (GreaterThan(key, rootKey)) // node.Key > root.Key
113             {
114                 root = GetRightOrDefault(root);
115             }
116             else // node.Key == root.Key
117             {
118                 return root;
119             }
120         }
121         return Zero;
122     }
123
124     // TODO: Return indices range instead of references count
125     [MethodImpl(MethodImplOptions.AggressiveInlining)]
126     public TLink CountUsages(TLink link) => GetSizeOrZero(GetTreeRoot(link));
127
128     [MethodImpl(MethodImplOptions.AggressiveInlining)]
129     public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
130         ↳ EachUsageCore(@base, GetTreeRoot(@base), handler);
131
132     // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
133         ↳ low-level MSIL stack.
134     [MethodImpl(MethodImplOptions.AggressiveInlining)]
135     private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
136     {

```

```

133     var @continue = Continue;
134     if (EqualToZero(link))
135     {
136         return @continue;
137     }
138     var @break = Break;
139     if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
140     {
141         return @break;
142     }
143     if (AreEqual(handler(GetLinkValues(link)), @break))
144     {
145         return @break;
146     }
147     if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
148     {
149         return @break;
150     }
151     return @continue;
152 }
153
154 [MethodImpl(MethodImplOptions.AggressiveInlining)]
155 protected override void PrintNodeValue(TLink node, StringBuilder sb)
156 {
157     ref var link = ref GetLinkDataPartReference(node);
158     sb.Append(' ');
159     sb.Append(link.Source);
160     sb.Append('-');
161     sb.Append('>');
162     sb.Append(link.Target);
163 }
164 }
165 }

```

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

```

```

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

```

### 1.39 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesLinkedListMethods.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Collections.Methods.Lists;
5 using Platform.Converters;
6 using static System.Runtime.CompilerServices.Unsafe;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Memory.Split.Generic
11 {
12     public unsafe class InternalLinksSourcesLinkedListMethods<TLink> :
        ↳ RelativeCircularDoublyLinkedListMethods<TLink>
13     {
14         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            ↳ UncheckedConverter<TLink, long>.Default;
15         private readonly byte* _linksDataParts;
16         private readonly byte* _linksIndexParts;
17         protected readonly TLink Break;
18         protected readonly TLink Continue;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

21 public InternalLinksSourcesLinkedListMethods(LinksConstants<TLink> constants, byte*
    ↳ linksDataParts, byte* linksIndexParts)
22 {
23     _linksDataParts = linksDataParts;
24     _linksIndexParts = linksIndexParts;
25     Break = constants.Break;
26     Continue = constants.Continue;
27 }
28
29 [MethodImpl(MethodImplOptions.AggressiveInlining)]
30 protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
    ↳ AsRef<RawLinkDataPart<TLink>>(_linksDataParts + (RawLinkDataPart<TLink>.SizeInBytes
    ↳ * _addressToInt64Converter.Convert(link)));
31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
    ↳ ref AsRef<RawLinkIndexPart<TLink>>(_linksIndexParts +
    ↳ (RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link)));
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 protected override TLink GetFirst(TLink head) =>
    ↳ GetLinkIndexPartReference(head).RootAsSource;
37
38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 protected override TLink GetLast(TLink head)
40 {
41     var first = GetLinkIndexPartReference(head).RootAsSource;
42     if (EqualToZero(first))
43     {
44         return first;
45     }
46     else
47     {
48         return GetPrevious(first);
49     }
50 }
51
52 [MethodImpl(MethodImplOptions.AggressiveInlining)]
53 protected override TLink GetPrevious(TLink element) =>
    ↳ GetLinkIndexPartReference(element).LeftAsSource;
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 protected override TLink GetNext(TLink element) =>
    ↳ GetLinkIndexPartReference(element).RightAsSource;
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 protected override TLink GetSize(TLink head) =>
    ↳ GetLinkIndexPartReference(head).SizeAsSource;
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected override void SetFirst(TLink head, TLink element) =>
    ↳ GetLinkIndexPartReference(head).RootAsSource = element;
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 protected override void SetLast(TLink head, TLink element)
66 {
67     //var first = GetLinkIndexPartReference(head).RootAsSource;
68     //if (EqualToZero(first))
69     //{
70         // SetFirst(head, element);
71     //}
72     //else
73     //{
74         // SetPrevious(first, element);
75     //}
76 }
77
78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 protected override void SetPrevious(TLink element, TLink previous) =>
    ↳ GetLinkIndexPartReference(element).LeftAsSource = previous;
80
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 protected override void SetNext(TLink element, TLink next) =>
    ↳ GetLinkIndexPartReference(element).RightAsSource = next;
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 protected override void SetSize(TLink head, TLink size) =>
    ↳ GetLinkIndexPartReference(head).SizeAsSource = size;

```



```

86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 public TLink CountUsages(TLink head) => GetSize(head);
88
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
91 {
92     ref var link = ref GetLinkDataPartReference(linkIndex);
93     return new Link<TLink>(linkIndex, link.Source, link.Target);
94 }
95
96 [MethodImpl(MethodImplOptions.AggressiveInlining)]
97 public TLink EachUsage(TLink source, Func<IList<TLink>, TLink> handler)
98 {
99     var @continue = Continue;
100     var @break = Break;
101     var current = GetFirst(source);
102     var first = current;
103     while (!EqualToZero(current))
104     {
105         if (AreEqual(handler(GetLinkValues(current)), @break))
106         {
107             return @break;
108         }
109         current = GetNext(current);
110         if (AreEqual(current, first))
111         {
112             return @continue;
113         }
114     }
115     return @continue;
116 }
117 }
118 }
119 }

```

#### 1.40 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesRecursionlessSizeBalancedTree

```

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 InternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLink> :
8     ↪ InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
12         ↪ constants, byte* linksDataParts, byte* linksIndexParts, byte* header) :
13         ↪ base(constants, 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)]

```

```

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

```

#### 1.41 ./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)]

```

```

37     protected override TLink GetTreeRoot(TLink link) =>
38         ↪ GetLinkIndexPartReference(link).RootAsSource;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override TLink GetBasePartValue(TLink link) =>
42         ↪ GetLinkDataPartReference(link).Source;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override TLink GetKeyPartValue(TLink link) =>
46         ↪ GetLinkDataPartReference(link).Target;
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected override void ClearNode(TLink node)
50     {
51         ref var link = ref GetLinkIndexPartReference(node);
52         link.LeftAsSource = Zero;
53         link.RightAsSource = Zero;
54         link.SizeAsSource = Zero;
55     }
56 }

```

## 1.42 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsRecursionlessSizeBalancedTree

```

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 InternalLinksTargetsRecursionlessSizeBalancedTreeMethods<TLink> :
8          ↪ InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
12             ↪ constants, byte* linksDataParts, byte* linksIndexParts, byte* header) :
13             ↪ base(constants, 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)]

```

```

40     protected override TLink GetBasePartValue(TLink link) =>
41         ↪ GetLinkDataPartReference(link).Target;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override TLink GetKeyPartValue(TLink link) =>
45         ↪ GetLinkDataPartReference(link).Source;
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override void ClearNode(TLink node)
49     {
50         ref var link = ref GetLinkIndexPartReference(node);
51         link.LeftAsTarget = Zero;
52         link.RightAsTarget = Zero;
53         link.SizeAsTarget = Zero;
54     }
55
56     public override TLink Search(TLink source, TLink target) =>
57         ↪ SearchCore(GetTreeRoot(target), source);
58 }

```

#### 1.43 ./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)]

```

```

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

```

#### 1.44 ./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(string dataMemory, string indexMemory) : this(new
23             ↪ FileMappedResizableDirectMemory(dataMemory), new
24             ↪ FileMappedResizableDirectMemory(indexMemory)) { }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
28             ↪ indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
32             ↪ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
33             ↪ memoryReservationStep, Default<LinksConstants<TLink>>.Instance,
34             ↪ IndexTreeType.Default, useLinkedList: true) { }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
38             ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
39             ↪ this(dataMemory, indexMemory, memoryReservationStep, constants,
40             ↪ IndexTreeType.Default, useLinkedList: true) { }
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
44             ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants,
45             ↪ IndexTreeType indexTreeType, bool useLinkedList) : base(dataMemory, indexMemory,
46             ↪ memoryReservationStep, constants, useLinkedList)
47         {
48             if (indexTreeType == IndexTreeType.SizeBalancedTree)
49             {
50                 _createInternalSourceTreeMethods = () => new
51                     ↪ InternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants,
52                     ↪ _linksDataParts, _linksIndexParts, _header);
53                 _createExternalSourceTreeMethods = () => new
54                     ↪ ExternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants,
55                     ↪ _linksDataParts, _linksIndexParts, _header);
56                 _createInternalTargetTreeMethods = () => new
57                     ↪ InternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants,
58                     ↪ _linksDataParts, _linksIndexParts, _header);
59                 _createExternalTargetTreeMethods = () => new
60                     ↪ ExternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants,
61                     ↪ _linksDataParts, _linksIndexParts, _header);
62             }
63         }
64     }
65 }

```

```

42     }
43     else
44     {
45         _createInternalSourceTreeMethods = () => new
46         ↪ InternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
47         ↪ _linksDataParts, _linksIndexParts, _header);
48         _createExternalSourceTreeMethods = () => new
49         ↪ ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
50         ↪ _linksDataParts, _linksIndexParts, _header);
51         _createInternalTargetTreeMethods = () => new
52         ↪ InternalLinksTargetsRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
53         ↪ _linksDataParts, _linksIndexParts, _header);
54         _createExternalTargetTreeMethods = () => new
55         ↪ ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
56         ↪ _linksDataParts, _linksIndexParts, _header);
57     }
58     Init(dataMemory, indexMemory);
59 }
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected override void SetPointers(IResizableDirectMemory dataMemory,
63 ↪ IResizableDirectMemory indexMemory)
64 {
65     _linksDataParts = (byte*)dataMemory.Pointer;
66     _linksIndexParts = (byte*)indexMemory.Pointer;
67     _header = _linksIndexParts;
68     if (_useLinkedList)
69     {
70         InternalSourcesListMethods = new
71         ↪ InternalLinksSourcesLinkedListMethods<TLink>(Constants, _linksDataParts,
72         ↪ _linksIndexParts);
73     }
74     else
75     {
76         InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
77     }
78     ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
79     InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
80     ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
81     UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_linksDataParts, _header);
82 }
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 protected override void ResetPointers()
86 {
87     base.ResetPointers();
88     _linksDataParts = null;
89     _linksIndexParts = null;
90     _header = null;
91 }
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 protected override ref LinksHeader<TLink> GetHeaderReference() => ref
95 ↪ AsRef<LinksHeader<TLink>>(_header);
96
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
99 ↪ => ref AsRef<RawLinkDataPart<TLink>>(_linksDataParts + (LinkDataPartSizeInBytes *
100 ↪ ConvertToInt64(linkIndex)));
101
102 [MethodImpl(MethodImplOptions.AggressiveInlining)]
103 protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
104 ↪ linkIndex) => ref AsRef<RawLinkIndexPart<TLink>>(_linksIndexParts +
105 ↪ (LinkIndexPartSizeInBytes * ConvertToInt64(linkIndex)));
106 }
107 }

```

#### 1.45 ./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 bool _useLinkedList;
45         protected readonly long _dataMemoryReservationStepInBytes;
46         protected readonly long _indexMemoryReservationStepInBytes;
47
48         protected InternalLinksSourcesLinkedListMethods<TLink> InternalSourcesListMethods;
49         protected ILinksTreeMethods<TLink> InternalSourcesTreeMethods;
50         protected ILinksTreeMethods<TLink> ExternalSourcesTreeMethods;
51         protected ILinksTreeMethods<TLink> InternalTargetsTreeMethods;
52         protected ILinksTreeMethods<TLink> ExternalTargetsTreeMethods;
53         // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
54         ↳ нужно использовать не список а дерево, так как так можно быстрее проверить на
55         ↳ наличие связи внутри
56         protected ILinksListMethods<TLink> UnusedLinksListMethods;
57
58         /// <summary>
59         ///     Возвращает общее число связей находящихся в хранилище.
60         /// </summary>
61         protected virtual TLink Total
62         {
63             [MethodImpl(MethodImplOptions.AggressiveInlining)]
64             get
65             {
66                 ref var header = ref GetHeaderReference();
67                 return Subtract(header.AllocatedLinks, header.FreeLinks);
68             }
69         }
70
71         public virtual LinksConstants<TLink> Constants
72         {
73             [MethodImpl(MethodImplOptions.AggressiveInlining)]
74             get;
75         }
76
77         [MethodImpl(MethodImplOptions.AggressiveInlining)]
78         protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
79             ↳ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants, bool
80             ↳ useLinkedList)
81         {
82             _dataMemory = dataMemory;
83             _indexMemory = indexMemory;
84             _dataMemoryReservationStepInBytes = memoryReservationStep * LinkDataPartSizeInBytes;
85             _indexMemoryReservationStepInBytes = memoryReservationStep *
86                 ↳ LinkIndexPartSizeInBytes;
87             _useLinkedList = useLinkedList;
88             Constants = constants;
89         }
90
91         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

83     protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
      ↪ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
      ↪ memoryReservationStep, Default<LinksConstants<TLink>>.Instance, useLinkedList: true)
      ↪ { }

84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     protected virtual void Init(IResizableDirectMemory dataMemory, IResizableDirectMemory
      ↪ indexMemory)
    {
87         // Read allocated links from header
88         if (indexMemory.ReservedCapacity < LinkHeaderSizeInBytes)
89         {
90             indexMemory.ReservedCapacity = LinkHeaderSizeInBytes;
91         }
92         SetPointers(dataMemory, indexMemory);
93         ref var header = ref GetHeaderReference();
94         var allocatedLinks = ConvertToInt64(header.AllocatedLinks);
95         // Adjust reserved capacity
96         var minimumDataReservedCapacity = allocatedLinks * LinkDataPartSizeInBytes;
97         if (minimumDataReservedCapacity < dataMemory.UsedCapacity)
98         {
99             minimumDataReservedCapacity = dataMemory.UsedCapacity;
100         }
101         if (minimumDataReservedCapacity < _dataMemoryReservationStepInBytes)
102         {
103             minimumDataReservedCapacity = _dataMemoryReservationStepInBytes;
104         }
105         var minimumIndexReservedCapacity = allocatedLinks * LinkDataPartSizeInBytes;
106         if (minimumIndexReservedCapacity < indexMemory.UsedCapacity)
107         {
108             minimumIndexReservedCapacity = indexMemory.UsedCapacity;
109         }
110         if (minimumIndexReservedCapacity < _indexMemoryReservationStepInBytes)
111         {
112             minimumIndexReservedCapacity = _indexMemoryReservationStepInBytes;
113         }
114         // Check for alignment
115         if (minimumDataReservedCapacity % _dataMemoryReservationStepInBytes > 0)
116         {
117             minimumDataReservedCapacity = ((minimumDataReservedCapacity /
      ↪ _dataMemoryReservationStepInBytes) * _dataMemoryReservationStepInBytes) +
      ↪ _dataMemoryReservationStepInBytes;
118         }
119         if (minimumIndexReservedCapacity % _indexMemoryReservationStepInBytes > 0)
120         {
121             minimumIndexReservedCapacity = ((minimumIndexReservedCapacity /
      ↪ _indexMemoryReservationStepInBytes) * _indexMemoryReservationStepInBytes) +
      ↪ _indexMemoryReservationStepInBytes;
122         }
123         if (dataMemory.ReservedCapacity != minimumDataReservedCapacity)
124         {
125             dataMemory.ReservedCapacity = minimumDataReservedCapacity;
126         }
127         if (indexMemory.ReservedCapacity != minimumIndexReservedCapacity)
128         {
129             indexMemory.ReservedCapacity = minimumIndexReservedCapacity;
130         }
131         SetPointers(dataMemory, indexMemory);
132         header = ref GetHeaderReference();
133         // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
134         // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
135         dataMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
      ↪ LinkDataPartSizeInBytes) + LinkDataPartSizeInBytes; // First link is read only
136         ↪ zero link.
137         indexMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
      ↪ LinkIndexPartSizeInBytes) + LinkHeaderSizeInBytes;
138         // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
139         // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
140         header.ReservedLinks = ConvertToAddress((dataMemory.ReservedCapacity -
      ↪ LinkDataPartSizeInBytes) / LinkDataPartSizeInBytes);
141     }
142
143     [MethodImpl(MethodImplOptions.AggressiveInlining)]
144     public virtual TLink Count(IList<TLink> restrictions)
    {
145         // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
146         if (restrictions.Count == 0)
147         {
148

```



```

149         return Total;
150     }
151     var constants = Constants;
152     var any = constants.Any;
153     var index = restrictions[constants.IndexPart];
154     if (restrictions.Count == 1)
155     {
156         if (AreEqual(index, any))
157         {
158             return Total;
159         }
160         return Exists(index) ? GetOne() : GetZero();
161     }
162     if (restrictions.Count == 2)
163     {
164         var value = restrictions[1];
165         if (AreEqual(index, any))
166         {
167             if (AreEqual(value, any))
168             {
169                 return Total; // Any - как отсутствие ограничения
170             }
171             var externalReferencesRange = constants.ExternalReferencesRange;
172             if (externalReferencesRange.HasValue &&
173                 ⇨ externalReferencesRange.Value.Contains(value))
174             {
175                 return Add(ExternalSourcesTreeMethods.CountUsages(value),
176                     ⇨ ExternalTargetsTreeMethods.CountUsages(value));
177             }
178             else
179             {
180                 if (_useLinkedList)
181                 {
182                     return Add(InternalSourcesListMethods.CountUsages(value),
183                         ⇨ InternalTargetsTreeMethods.CountUsages(value));
184                 }
185                 else
186                 {
187                     return Add(InternalSourcesTreeMethods.CountUsages(value),
188                         ⇨ InternalTargetsTreeMethods.CountUsages(value));
189                 }
190             }
191         }
192         else
193         {
194             if (!Exists(index))
195             {
196                 return GetZero();
197             }
198             if (AreEqual(value, any))
199             {
200                 return GetOne();
201             }
202             ref var storedLinkValue = ref GetLinkDataPartReference(index);
203             if (AreEqual(storedLinkValue.Source, value) ||
204                 ⇨ AreEqual(storedLinkValue.Target, value))
205             {
206                 return GetOne();
207             }
208             return GetZero();
209         }
210     }
211     if (restrictions.Count == 3)
212     {
213         var externalReferencesRange = constants.ExternalReferencesRange;
214         var source = restrictions[constants.SourcePart];
215         var target = restrictions[constants.TargetPart];
216         if (AreEqual(index, any))
217         {
218             if (AreEqual(source, any) && AreEqual(target, any))
219             {
220                 return Total;
221             }
222             else if (AreEqual(source, any))
223             {
224                 if (externalReferencesRange.HasValue &&
225                     ⇨ externalReferencesRange.Value.Contains(target))
226                 {

```

```

221         return ExternalTargetsTreeMethods.CountUsages(target);
222     }
223     else
224     {
225         return InternalTargetsTreeMethods.CountUsages(target);
226     }
227 }
228 else if (AreEqual(target, any))
229 {
230     if (externalReferencesRange.HasValue &&
231         ↪ externalReferencesRange.Value.Contains(source))
232     {
233         return ExternalSourcesTreeMethods.CountUsages(source);
234     }
235     else
236     {
237         if (_useLinkedList)
238         {
239             return InternalSourcesListMethods.CountUsages(source);
240         }
241         else
242         {
243             return InternalSourcesTreeMethods.CountUsages(source);
244         }
245     }
246 }
247 else //if(source != Any && target != Any)
248 {
249     // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
250     TLink link;
251     if (externalReferencesRange.HasValue)
252     {
253         if (externalReferencesRange.Value.Contains(source) &&
254             ↪ externalReferencesRange.Value.Contains(target))
255         {
256             link = ExternalSourcesTreeMethods.Search(source, target);
257         }
258         else if (externalReferencesRange.Value.Contains(source))
259         {
260             link = InternalTargetsTreeMethods.Search(source, target);
261         }
262         else if (externalReferencesRange.Value.Contains(target))
263         {
264             if (_useLinkedList)
265             {
266                 link = ExternalSourcesTreeMethods.Search(source, target);
267             }
268             else
269             {
270                 link = InternalSourcesTreeMethods.Search(source, target);
271             }
272         }
273         else
274         {
275             if (_useLinkedList ||
276                 ↪ GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
277                 ↪ InternalTargetsTreeMethods.CountUsages(target)))
278             {
279                 link = InternalTargetsTreeMethods.Search(source, target);
280             }
281             else
282             {
283                 link = InternalSourcesTreeMethods.Search(source, target);
284             }
285         }
286     }
287     else
288     {
289         if (_useLinkedList ||
290             ↪ GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
291             ↪ InternalTargetsTreeMethods.CountUsages(target)))
292         {
293             link = InternalTargetsTreeMethods.Search(source, target);
294         }
295         else
296         {
297             link = InternalSourcesTreeMethods.Search(source, target);
298         }
299     }

```

```

293     }
294     return AreEqual(link, constants.Null) ? GetZero() : GetOne();
295 }
296 }
297 else
298 {
299     if (!Exists(index))
300     {
301         return GetZero();
302     }
303     if (AreEqual(source, any) && AreEqual(target, any))
304     {
305         return GetOne();
306     }
307     ref var storedLinkValue = ref GetLinkDataPartReference(index);
308     if (!AreEqual(source, any) && !AreEqual(target, any))
309     {
310         if (AreEqual(storedLinkValue.Source, source) &&
311             ↪ AreEqual(storedLinkValue.Target, target))
312         {
313             return GetOne();
314         }
315         return GetZero();
316     }
317     var value = default(TLink);
318     if (AreEqual(source, any))
319     {
320         value = target;
321     }
322     if (AreEqual(target, any))
323     {
324         value = source;
325     }
326     if (AreEqual(storedLinkValue.Source, value) ||
327         ↪ AreEqual(storedLinkValue.Target, value))
328     {
329         return GetOne();
330     }
331     return GetZero();
332 }
333 }
334
335 [MethodImpl(MethodImplOptions.AggressiveInlining)]
336 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
337 {
338     var constants = Constants;
339     var @break = constants.Break;
340     if (restrictions.Count == 0)
341     {
342         for (var link = GetOne(); LessOrEqualThan(link,
343             ↪ GetHeaderReference().AllocatedLinks); link = Increment(link))
344         {
345             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
346             {
347                 return @break;
348             }
349         }
350         return @break;
351     }
352     var @continue = constants.Continue;
353     var any = constants.Any;
354     var index = restrictions[constants.IndexPart];
355     if (restrictions.Count == 1)
356     {
357         if (AreEqual(index, any))
358         {
359             return Each(handler, Array.Empty<TLink>());
360         }
361         if (!Exists(index))
362         {
363             return @continue;
364         }
365         return handler(GetLinkStruct(index));
366     }
367     if (restrictions.Count == 2)

```

```

367 {
368     var value = restrictions[1];
369     if (AreEqual(index, any))
370     {
371         if (AreEqual(value, any))
372         {
373             return Each(handler, Array.Empty<TLink>());
374         }
375         if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
376         {
377             return @break;
378         }
379         return Each(handler, new Link<TLink>(index, any, value));
380     }
381     else
382     {
383         if (!Exists(index))
384         {
385             return @continue;
386         }
387         if (AreEqual(value, any))
388         {
389             return handler(GetLinkStruct(index));
390         }
391         ref var storedLinkValue = ref GetLinkDataPartReference(index);
392         if (AreEqual(storedLinkValue.Source, value) ||
393             AreEqual(storedLinkValue.Target, value))
394         {
395             return handler(GetLinkStruct(index));
396         }
397         return @continue;
398     }
399 }
400 if (restrictions.Count == 3)
401 {
402     var externalReferencesRange = constants.ExternalReferencesRange;
403     var source = restrictions[constants.SourcePart];
404     var target = restrictions[constants.TargetPart];
405     if (AreEqual(index, any))
406     {
407         if (AreEqual(source, any) && AreEqual(target, any))
408         {
409             return Each(handler, Array.Empty<TLink>());
410         }
411         else if (AreEqual(source, any))
412         {
413             if (externalReferencesRange.HasValue &&
414                 ⇨ externalReferencesRange.Value.Contains(target))
415             {
416                 return ExternalTargetsTreeMethods.EachUsage(target, handler);
417             }
418             else
419             {
420                 return InternalTargetsTreeMethods.EachUsage(target, handler);
421             }
422         }
423         else if (AreEqual(target, any))
424         {
425             if (externalReferencesRange.HasValue &&
426                 ⇨ externalReferencesRange.Value.Contains(source))
427             {
428                 return ExternalSourcesTreeMethods.EachUsage(source, handler);
429             }
430             else
431             {
432                 if (_useLinkedList)
433                 {
434                     return InternalSourcesListMethods.EachUsage(source, handler);
435                 }
436                 else
437                 {
438                     return InternalSourcesTreeMethods.EachUsage(source, handler);
439                 }
440             }
441         }
442         else //if(source != Any && target != Any)
443         {
444             TLink link;

```

```

443     if (externalReferencesRange.HasValue)
444     {
445         if (externalReferencesRange.Value.Contains(source) &&
446             ↪ externalReferencesRange.Value.Contains(target))
447         {
448             link = ExternalSourcesTreeMethods.Search(source, target);
449         }
450         else if (externalReferencesRange.Value.Contains(source))
451         {
452             link = InternalTargetsTreeMethods.Search(source, target);
453         }
454         else if (externalReferencesRange.Value.Contains(target))
455         {
456             if (_useLinkedList)
457             {
458                 link = ExternalSourcesTreeMethods.Search(source, target);
459             }
460             else
461             {
462                 link = InternalSourcesTreeMethods.Search(source, target);
463             }
464         }
465         else
466         {
467             if (_useLinkedList ||
468                 ↪ GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
469                 ↪ InternalTargetsTreeMethods.CountUsages(target)))
470             {
471                 link = InternalTargetsTreeMethods.Search(source, target);
472             }
473             else
474             {
475                 link = InternalSourcesTreeMethods.Search(source, target);
476             }
477         }
478     }
479     }
480     }
481     else
482     {
483         if (_useLinkedList ||
484             ↪ GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
485             ↪ InternalTargetsTreeMethods.CountUsages(target)))
486         {
487             link = InternalTargetsTreeMethods.Search(source, target);
488         }
489         else
490         {
491             link = InternalSourcesTreeMethods.Search(source, target);
492         }
493     }
494     return AreEqual(link, constants.Null) ? @continue :
495         ↪ handler(GetLinkStruct(link));
496 }
497 }
498 else
499 {
500     if (!Exists(index))
501     {
502         return @continue;
503     }
504     if (AreEqual(source, any) && AreEqual(target, any))
505     {
506         return handler(GetLinkStruct(index));
507     }
508     ref var storedLinkValue = ref GetLinkDataPartReference(index);
509     if (!AreEqual(source, any) && !AreEqual(target, any))
510     {
511         if (AreEqual(storedLinkValue.Source, source) &&
512             AreEqual(storedLinkValue.Target, target))
513         {
514             return handler(GetLinkStruct(index));
515         }
516         return @continue;
517     }
518     var value = default(TLink);
519     if (AreEqual(source, any))
520     {
521         value = target;
522     }

```

```

515         if (AreEqual(target, any))
516         {
517             value = source;
518         }
519         if (AreEqual(storedLinkValue.Source, value) ||
520             AreEqual(storedLinkValue.Target, value))
521         {
522             return handler(GetLinkStruct(index));
523         }
524         return @continue;
525     }
526 }
527 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
528 }
529
530 /// <remarks>
531 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↳ в другом месте (но не в менеджере памяти, а в логике Links)
532 /// </remarks>
533 [MethodImpl(MethodImplOptions.AggressiveInlining)]
534 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
535 {
536     var constants = Constants;
537     var @null = constants.Null;
538     var externalReferencesRange = constants.ExternalReferencesRange;
539     var linkIndex = restrictions[constants.IndexPart];
540     ref var link = ref GetLinkDataPartReference(linkIndex);
541     var source = link.Source;
542     var target = link.Target;
543     ref var header = ref GetHeaderReference();
544     ref var rootAsSource = ref header.RootAsSource;
545     ref var rootAsTarget = ref header.RootAsTarget;
546     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↳ предварительно заполнено нулями
547     if (!AreEqual(source, @null))
548     {
549         if (externalReferencesRange.HasValue &&
550             ↳ externalReferencesRange.Value.Contains(source))
551         {
552             ExternalSourcesTreeMethods.Detach(ref rootAsSource, linkIndex);
553         }
554         else
555         {
556             if (_useLinkedList)
557             {
558                 InternalSourcesListMethods.Detach(source, linkIndex);
559             }
560             else
561             {
562                 InternalSourcesTreeMethods.Detach(ref
563                     ↳ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
564             }
565         }
566     }
567     if (!AreEqual(target, @null))
568     {
569         if (externalReferencesRange.HasValue &&
570             ↳ externalReferencesRange.Value.Contains(target))
571         {
572             ExternalTargetsTreeMethods.Detach(ref rootAsTarget, linkIndex);
573         }
574         else
575         {
576             InternalTargetsTreeMethods.Detach(ref
577                 ↳ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
578         }
579     }
580     source = link.Source = substitution[constants.SourcePart];
581     target = link.Target = substitution[constants.TargetPart];
582     if (!AreEqual(source, @null))
583     {
584         if (externalReferencesRange.HasValue &&
585             ↳ externalReferencesRange.Value.Contains(source))
586         {
587             ExternalSourcesTreeMethods.Attach(ref rootAsSource, linkIndex);
588         }
589         else
590         {
591             InternalSourcesListMethods.Attach(source, linkIndex);
592         }
593     }
594     if (!AreEqual(target, @null))
595     {
596         if (externalTargetsRange.HasValue &&
597             ↳ externalTargetsRange.Value.Contains(target))
598         {
599             ExternalTargetsTreeMethods.Attach(ref rootAsTarget, linkIndex);
600         }
601         else
602         {
603             InternalTargetsListMethods.Attach(target, linkIndex);
604         }
605     }
606 }

```

```

585     {
586         if (_useLinkedList)
587         {
588             InternalSourcesListMethods.AttachAsLast(source, linkIndex);
589         }
590         else
591         {
592             InternalSourcesTreeMethods.Attach(ref
593                 ↪ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
594         }
595     }
596     if (!AreEqual(target, @null))
597     {
598         if (externalReferencesRange.HasValue &&
599             ↪ externalReferencesRange.Value.Contains(target))
600         {
601             ExternalTargetsTreeMethods.Attach(ref rootAsTarget, linkIndex);
602         }
603         else
604         {
605             InternalTargetsTreeMethods.Attach(ref
606                 ↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
607         }
608     }
609     return linkIndex;
610 }
611
612 /// <remarks>
613 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
614 ↪ пространство
615 /// </remarks>
616 [MethodImpl(MethodImplOptions.AggressiveInlining)]
617 public virtual TLink Create(IList<TLink> restrictions)
618 {
619     ref var header = ref GetHeaderReference();
620     var freeLink = header.FirstFreeLink;
621     if (!AreEqual(freeLink, Constants.Null))
622     {
623         UnusedLinksListMethods.Detach(freeLink);
624     }
625     else
626     {
627         var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
628         if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
629         {
630             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
631         }
632         if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
633         {
634             _dataMemory.ReservedCapacity += _dataMemory.ReservationStepInBytes;
635             _indexMemory.ReservedCapacity += _indexMemory.ReservationStepInBytes;
636             SetPointers(_dataMemory, _indexMemory);
637             header = ref GetHeaderReference();
638             header.ReservedLinks = ConvertToAddress(_dataMemory.ReservedCapacity /
639                 ↪ LinkDataPartSizeInBytes);
640         }
641         freeLink = header.AllocatedLinks = Increment(header.AllocatedLinks);
642         _dataMemory.UsedCapacity += LinkDataPartSizeInBytes;
643         _indexMemory.UsedCapacity += LinkIndexPartSizeInBytes;
644     }
645     return freeLink;
646 }
647
648 [MethodImpl(MethodImplOptions.AggressiveInlining)]
649 public virtual void Delete(IList<TLink> restrictions)
650 {
651     ref var header = ref GetHeaderReference();
652     var link = restrictions[Constants.IndexPart];
653     if (LessThan(link, header.AllocatedLinks)
654     {
655         UnusedLinksListMethods.AttachAsFirst(link);
656     }
657     else if (AreEqual(link, header.AllocatedLinks))
658     {
659         header.AllocatedLinks = Decrement(header.AllocatedLinks);
660         _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
661         _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
662     }
663 }

```

```

658         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
659         ↪ пока не дойдём до первой существующей связи
660         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
661         while (GreaterThan(header.AllocatedLinks, GetZero()) &&
662             ↪ IsUnusedLink(header.AllocatedLinks))
663         {
664             UnusedLinksListMethods.Detach(header.AllocatedLinks);
665             header.AllocatedLinks = Decrement(header.AllocatedLinks);
666             _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
667             _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
668         }
669     }
670
671     [MethodImpl(MethodImplOptions.AggressiveInlining)]
672     public IList<TLink> GetLinkStruct(TLink linkIndex)
673     {
674         ref var link = ref GetLinkDataPartReference(linkIndex);
675         return new Link<TLink>(linkIndex, link.Source, link.Target);
676     }
677
678     /// <remarks>
679     /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
680     ↪ адрес реально поменялся
681     ///
682     /// Указатель this.links может быть в том же месте,
683     /// так как 0-я связь не используется и имеет такой же размер как Header,
684     /// поэтому header размещается в том же месте, что и 0-я связь
685     /// </remarks>
686     [MethodImpl(MethodImplOptions.AggressiveInlining)]
687     protected abstract void SetPointers(IResizableDirectMemory dataMemory,
688         ↪ IResizableDirectMemory indexMemory);
689
690     [MethodImpl(MethodImplOptions.AggressiveInlining)]
691     protected virtual void ResetPointers()
692     {
693         InternalSourcesListMethods = null;
694         InternalSourcesTreeMethods = null;
695         ExternalSourcesTreeMethods = null;
696         InternalTargetsTreeMethods = null;
697         ExternalTargetsTreeMethods = null;
698         UnusedLinksListMethods = null;
699     }
700
701     [MethodImpl(MethodImplOptions.AggressiveInlining)]
702     protected abstract ref LinksHeader<TLink> GetHeaderReference();
703
704     [MethodImpl(MethodImplOptions.AggressiveInlining)]
705     protected abstract ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex);
706
707     [MethodImpl(MethodImplOptions.AggressiveInlining)]
708     protected abstract ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
709         ↪ linkIndex);
710
711     [MethodImpl(MethodImplOptions.AggressiveInlining)]
712     protected virtual bool Exists(TLink link)
713     => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
714         && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
715         && !IsUnusedLink(link);
716
717     [MethodImpl(MethodImplOptions.AggressiveInlining)]
718     protected virtual bool IsUnusedLink(TLink linkIndex)
719     {
720         if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
721         ↪ is not needed
722         {
723             // TODO: Reduce access to memory in different location (should be enough to use
724             ↪ just linkIndexPart)
725             ref var linkDataPart = ref GetLinkDataPartReference(linkIndex);
726             ref var linkIndexPart = ref GetLinkIndexPartReference(linkIndex);
727             return AreEqual(linkIndexPart.SizeAsTarget, default) &&
728                 ↪ !AreEqual(linkDataPart.Source, default);
729         }
730         else
731         {
732             return true;
733         }
734     }
735 }

```



```

729 [MethodImpl(MethodImplOptions.AggressiveInlining)]
730 protected virtual TLink GetOne() => _one;
731
732 [MethodImpl(MethodImplOptions.AggressiveInlining)]
733 protected virtual TLink GetZero() => default;
734
735 [MethodImpl(MethodImplOptions.AggressiveInlining)]
736 protected virtual bool AreEqual(TLink first, TLink second) =>
    ↳ _equalityComparer.Equals(first, second);
737
738 [MethodImpl(MethodImplOptions.AggressiveInlining)]
739 protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
    ↳ second) < 0;
740
741 [MethodImpl(MethodImplOptions.AggressiveInlining)]
742 protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
    ↳ _comparer.Compare(first, second) <= 0;
743
744 [MethodImpl(MethodImplOptions.AggressiveInlining)]
745 protected virtual bool GreaterThan(TLink first, TLink second) =>
    ↳ _comparer.Compare(first, second) > 0;
746
747 [MethodImpl(MethodImplOptions.AggressiveInlining)]
748 protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
    ↳ _comparer.Compare(first, second) >= 0;
749
750 [MethodImpl(MethodImplOptions.AggressiveInlining)]
751 protected virtual long ConvertToInt64(TLink value) =>
    ↳ _addressToInt64Converter.Convert(value);
752
753 [MethodImpl(MethodImplOptions.AggressiveInlining)]
754 protected virtual TLink ConvertToAddress(long value) =>
    ↳ _int64ToAddressConverter.Convert(value);
755
756 [MethodImpl(MethodImplOptions.AggressiveInlining)]
757 protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
    ↳ second);
758
759 [MethodImpl(MethodImplOptions.AggressiveInlining)]
760 protected virtual TLink Subtract(TLink first, TLink second) =>
    ↳ Arithmetic<TLink>.Subtract(first, second);
761
762 [MethodImpl(MethodImplOptions.AggressiveInlining)]
763 protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
764
765 [MethodImpl(MethodImplOptions.AggressiveInlining)]
766 protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
767
768 #region Disposable
769
770 protected override bool AllowMultipleDisposeCalls
771 {
772     [MethodImpl(MethodImplOptions.AggressiveInlining)]
773     get => true;
774 }
775
776 [MethodImpl(MethodImplOptions.AggressiveInlining)]
777 protected override void Dispose(bool manual, bool wasDisposed)
778 {
779     if (!wasDisposed)
780     {
781         ResetPointers();
782         _dataMemory.DisposeIfPossible();
783         _indexMemory.DisposeIfPossible();
784     }
785 }
786
787 #endregion
788 }
789 }

```

#### 1.46 ./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> :
11         ↳ AbsoluteCircularDoublyLinkedListMethods<TLink>, ILinksListMethods<TLink>
12     {
13         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
14             ↳ UncheckedConverter<TLink, long>.Default;
15
16         private readonly byte* _links;
17         private readonly byte* _header;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnusedLinksListMethods(byte* links, byte* header)
21         {
22             _links = links;
23             _header = header;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
28             ↳ AsRef<LinksHeader<TLink>>(_header);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
32             ↳ AsRef<RawLinkDataPart<TLink>>(_links + (RawLinkDataPart<TLink>.SizeInBytes *
33             ↳ _addressToInt64Converter.Convert(link)));
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected override TLink GetPrevious(TLink element) =>
43             ↳ GetLinkDataPartReference(element).Source;
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override TLink GetNext(TLink element) =>
47             ↳ GetLinkDataPartReference(element).Target;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override TLink GetSize() => GetHeaderReference().FreeLinks;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
54             ↳ element;
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
58             ↳ element;
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         protected override void SetPrevious(TLink element, TLink previous) =>
62             ↳ GetLinkDataPartReference(element).Source = previous;
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected override void SetNext(TLink element, TLink next) =>
66             ↳ GetLinkDataPartReference(element).Target = next;
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
70     }
71 }

```

#### 1.47 ./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 }

```

```

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

```

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

```

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

```

```
48 }
```

#### 1.49 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksRecursionlessSizeBalancedTree

```
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 UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase :
10         ↳ ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<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
18         ↳ UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
19         ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
20         ↳ linksIndexParts, LinksHeader<TLink>* header)
21         : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
22     {
23         LinksDataParts = linksDataParts;
24         LinksIndexParts = linksIndexParts;
25         Header = header;
26     }
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     protected override TLink GetZero() => 0U;
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override bool EqualToZero(TLink value) => value == 0U;
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected override bool AreEqual(TLink first, TLink second) => first == second;
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override bool GreaterThanZero(TLink value) => value > 0U;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override bool GreaterThan(TLink first, TLink second) => first > second;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override bool GreaterOrEqualThan(TLink first, TLink second) => first >= second;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override bool GreaterOrEqualThanZero(TLink value) => true; // value >= 0 is
48     ↳ always true for ulong
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override bool LessOrEqualThanZero(TLink value) => value == 0UL; // value is
52     ↳ always >= 0 for ulong
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override TLink Increment(TLink value) => ++value;
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected override TLink Decrement(TLink value) => --value;
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override TLink Add(TLink first, TLink second) => first + second;
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override TLink Subtract(TLink first, TLink second) => first - second;
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override ref LinksHeader<TLink> GetHeaderReference() => ref *Header;
```

```

71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
73         ↪ ref LinksDataParts[link];
74
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
77         ↪ ref LinksIndexParts[link];
78
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     protected override bool FirstIsToLeftOfSecond(TLink first, TLink second)
81     {
82         ref var firstLink = ref LinksDataParts[first];
83         ref var secondLink = ref LinksDataParts[second];
84         return FirstIsToLeftOfSecond(firstLink.Source, firstLink.Target,
85             ↪ secondLink.Source, secondLink.Target);
86     }
87
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
90     {
91         ref var firstLink = ref LinksDataParts[first];
92         ref var secondLink = ref LinksDataParts[second];
93         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
94             ↪ secondLink.Source, secondLink.Target);
95     }
96 }

```

## 1.50 ./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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override bool AreEqual(TLink first, TLink second) => first == second;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override bool GreaterThanZero(TLink value) => value > 0U;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override bool GreaterThan(TLink first, TLink second) => first > second;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override bool GreaterOrEqualThan(TLink first, TLink second) => first >= second;
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override bool GreaterOrEqualThanZero(TLink value) => true; // value >= 0 is
47             ↪ always true for ulong
48     }
49 }

```

```

46     protected override bool LessOrEqualThanZero(TLink value) => value == 0UL; // value is
    ↪ always >= 0 for ulong
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override bool LessThanZero(TLink value) => false; // value < 0 is always false
    ↪ for ulong
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected override bool LessThan(TLink first, TLink second) => first < second;
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override TLink Increment(TLink value) => ++value;
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected override TLink Decrement(TLink value) => --value;
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override TLink Add(TLink first, TLink second) => first + second;
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override TLink Subtract(TLink first, TLink second) => first - second;
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override ref LinksHeader<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 FirstIsToTheLeftOfSecond(TLink first, TLink second)
80     {
81         ref var firstLink = ref LinksDataParts[first];
82         ref var secondLink = ref LinksDataParts[second];
83         return FirstIsToTheLeftOfSecond(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,
    ↪ secondLink.Source, secondLink.Target);
92     }
93 }
94 }

```

## 1.51 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSourcesRecursionlessSizeBalance

```

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 UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
    ↪ UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public
    ↪ UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(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].LeftAsSource;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

17     protected override ref TLink GetRightReference(TLink node) => ref
18         ↳ LinksIndexParts[node].RightAsSource;
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected override void SetLeft(TLink node, TLink left) =>
28         ↳ LinksIndexParts[node].LeftAsSource = left;
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     protected override void SetRight(TLink node, TLink right) =>
32         ↳ LinksIndexParts[node].RightAsSource = right;
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override void SetSize(TLink node, TLink size) =>
39         ↳ LinksIndexParts[node].SizeAsSource = size;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override TLink GetTreeRoot() => Header->RootAsSource;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
49         ↳ TLink secondSource, TLink secondTarget)
50         => firstSource < secondSource || firstSource == secondSource && firstTarget <
51             ↳ secondTarget;
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
55         ↳ TLink secondSource, TLink secondTarget)
56         => firstSource > secondSource || firstSource == secondSource && firstTarget >
57             ↳ secondTarget;
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override void ClearNode(TLink node)
61     {
62         ref var link = ref LinksIndexParts[node];
63         link.LeftAsSource = Zero;
64         link.RightAsSource = Zero;
65         link.SizeAsSource = Zero;
66     }
67 }

```

## 1.52 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSourcesSizeBalancedTreeMethods

```

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)]

```

```

20     protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     protected override void SetLeft(TLink node, TLink left) =>
27         ↳ LinksIndexParts[node].LeftAsSource = left;
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     protected override void SetRight(TLink node, TLink right) =>
31         ↳ LinksIndexParts[node].RightAsSource = right;
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected override void SetSize(TLink node, TLink size) =>
38         ↳ LinksIndexParts[node].SizeAsSource = size;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override TLink GetTreeRoot() => Header->RootAsSource;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
48         ↳ TLink secondSource, TLink secondTarget)
49         => firstSource < secondSource || firstSource == secondSource && firstTarget <
50             ↳ secondTarget;
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected override bool FirstIsToTheRightOfSecond(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 void ClearNode(TLink node)
60     {
61         ref var link = ref LinksIndexParts[node];
62         link.LeftAsSource = Zero;
63         link.RightAsSource = Zero;
64         link.SizeAsSource = Zero;
65     }
66 }

```

### 1.53 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksTargetsRecursionlessSizeBalance

```

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 UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
9          ↳ UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public
13         ↳ UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
14             ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
15             ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
16                 ↳ linksIndexParts, header) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref TLink GetLeftReference(TLink node) => ref
20             ↳ LinksIndexParts[node].LeftAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ref TLink GetRightReference(TLink node) => ref
24             ↳ LinksIndexParts[node].RightAsTarget;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsTarget;
28
29         [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) =>
27         ↳ LinksIndexParts[node].LeftAsTarget = left;
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     protected override void SetRight(TLink node, TLink right) =>
31         ↳ LinksIndexParts[node].RightAsTarget = right;
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected override void SetSize(TLink node, TLink size) =>
38         ↳ LinksIndexParts[node].SizeAsTarget = size;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override TLink GetTreeRoot() => Header->RootAsTarget;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
48         ↳ TLink secondSource, TLink secondTarget)
49         => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
50             ↳ secondSource;
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected override bool FirstIsToTheRightOfSecond(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 void ClearNode(TLink node)
60     {
61         ref var link = ref LinksIndexParts[node];
62         link.LeftAsTarget = Zero;
63         link.RightAsTarget = Zero;
64         link.SizeAsTarget = Zero;
65     }
66 }

```

#### 1.54 ./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
10      {
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          public UInt32ExternalLinksTargetsSizeBalancedTreeMethods(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  }

```

```

26     protected override void SetLeft(TLink node, TLink left) =>
27         ↳ LinksIndexParts[node].LeftAsTarget = left;
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     protected override void SetRight(TLink node, TLink right) =>
31         ↳ LinksIndexParts[node].RightAsTarget = right;
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected override void SetSize(TLink node, TLink size) =>
38         ↳ LinksIndexParts[node].SizeAsTarget = size;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override TLink GetTreeRoot() => Header->RootAsTarget;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
48         ↳ TLink secondSource, TLink secondTarget)
49         => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
50         ↳ secondSource;
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected override bool FirstIsToTheRightOfSecond(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 void ClearNode(TLink node)
60     {
61         ref var link = ref LinksIndexParts[node];
62         link.LeftAsTarget = Zero;
63         link.RightAsTarget = Zero;
64         link.SizeAsTarget = Zero;
65     }
66 }

```

## 1.55 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksRecursionlessSizeBalancedTreeM

```

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 UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase :
10         ↳ InternalLinksRecursionlessSizeBalancedTreeMethodsBase<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
18             ↳ UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
19             ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
20             ↳ linksIndexParts, LinksHeader<TLink>* header)
21             : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
22          {
23              LinksDataParts = linksDataParts;
24              LinksIndexParts = linksIndexParts;
25              Header = header;
26          }
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected override TLink GetZero() => 0U;
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          protected override bool EqualToZero(TLink value) => value == 0U;
33
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          protected override bool AreEqual(TLink first, TLink second) => first == second;
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

```

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 RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
74     ↪ ref LinksDataParts[link];
75
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
78     ↪ ref LinksIndexParts[link];
79
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     protected override bool FirstIsToLeftOfSecond(TLink first, TLink second) =>
82     ↪ GetKeyPartValue(first) < GetKeyPartValue(second);
83
84     [MethodImpl(MethodImplOptions.AggressiveInlining)]
85     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
86     ↪ GetKeyPartValue(first) > GetKeyPartValue(second);
87
88 }
89
90 }

```

## 1.56 ./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 :
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 UInt32InternalLinksSizeBalancedTreeMethodsBase(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         }
24     }
25 }

```

```

20     LinksIndexParts = linksIndexParts;
21     Header = header;
22 }
23
24 [MethodImpl(MethodImplOptions.AggressiveInlining)]
25 protected override TLink GetZero() => OU;
26
27 [MethodImpl(MethodImplOptions.AggressiveInlining)]
28 protected override bool EqualToZero(TLink value) => value == OU;
29
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 > OU;
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
    ↳ always true for ulong
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 protected override bool LessOrEqualThanZero(TLink value) => value == OUL; // value is
    ↳ always >= 0 for ulong
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override bool LessThanZero(TLink value) => false; // value < 0 is always false
    ↳ for ulong
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected override bool LessThan(TLink first, TLink second) => first < second;
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 protected override TLink Increment(TLink value) => ++value;
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected override TLink Decrement(TLink value) => --value;
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected override TLink Add(TLink first, TLink second) => first + second;
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 protected override TLink Subtract(TLink first, TLink second) => first - second;
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
    ↳ ref LinksDataParts[link];
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
    ↳ ref LinksIndexParts[link];
74
75 [MethodImpl(MethodImplOptions.AggressiveInlining)]
76 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
    ↳ GetKeyPartValue(first) < GetKeyPartValue(second);
77
78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
    ↳ GetKeyPartValue(first) > GetKeyPartValue(second);
80 }
81 }

```

## 1.57 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesLinkedListMethods.cs

```

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.Generic
7 {
8     public unsafe class UInt32InternalLinksSourcesLinkedListMethods :
9         ↳ InternalLinksSourcesLinkedListMethods<TLink>

```

```

9 {
10     private readonly RawLinkDataPart<TLink>* _linksDataParts;
11     private readonly RawLinkIndexPart<TLink>* _linksIndexParts;
12
13     [MethodImpl(MethodImplOptions.AggressiveInlining)]
14     public UInt32InternalLinksSourcesLinkedListMethods(LinksConstants<TLink> constants,
15         ↳ RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>* linksIndexParts)
16         : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts)
17     {
18         _linksDataParts = linksDataParts;
19         _linksIndexParts = linksIndexParts;
20     }
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
24         ↳ ref _linksDataParts[link];
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
28         ↳ ref _linksIndexParts[link];
29 }

```

## 1.58 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods

```

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 UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
9         ↳ UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public
13         ↳ UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
14         ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
15         ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
16         ↳ linksIndexParts, header) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref TLink GetLeftReference(TLink node) => ref
20         ↳ LinksIndexParts[node].LeftAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ref TLink GetRightReference(TLink node) => ref
24         ↳ LinksIndexParts[node].RightAsSource;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetLeft(TLink node, TLink left) =>
34         ↳ LinksIndexParts[node].LeftAsSource = left;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override void SetRight(TLink node, TLink right) =>
38         ↳ LinksIndexParts[node].RightAsSource = right;
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override void SetSize(TLink node, TLink size) =>
45         ↳ LinksIndexParts[node].SizeAsSource = size;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override TLink GetTreeRoot(TLink node) => LinksIndexParts[node].RootAsSource;
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Target;
55     }
56 }

```

```

46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override void ClearNode(TLink node)
48     {
49         ref var link = ref LinksIndexParts[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 }

```

## 1.59 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesSizeBalancedTreeMethods

```

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 }

```

```

56     }
57 }

```

## 1.60 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksTargetsRecursionlessSizeBalancedTreeMethods

```

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 UInt32InternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
9      ↪ UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public
13         ↪ UInt32InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
14         ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
15         ↪ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
16         ↪ linksIndexParts, header) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref TLink GetLeftReference(TLink node) => ref
20         ↪ LinksIndexParts[node].LeftAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ref TLink GetRightReference(TLink node) => ref
24         ↪ LinksIndexParts[node].RightAsTarget;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsTarget;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsTarget;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetLeft(TLink node, TLink left) =>
34         ↪ LinksIndexParts[node].LeftAsTarget = left;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override void SetRight(TLink node, TLink right) =>
38         ↪ LinksIndexParts[node].RightAsTarget = right;
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override void SetSize(TLink node, TLink size) =>
45         ↪ LinksIndexParts[node].SizeAsTarget = size;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override TLink GetTreeRoot(TLink node) => LinksIndexParts[node].RootAsTarget;
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Source;
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override void ClearNode(TLink node)
58         {
59             ref var link = ref LinksIndexParts[node];
60             link.LeftAsTarget = Zero;
61             link.RightAsTarget = Zero;
62             link.SizeAsTarget = Zero;
63         }
64
65         public override TLink Search(TLink source, TLink target) =>
66         ↪ SearchCore(GetTreeRoot(target), source);
67     }
68 }

```

## 1.61 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksTargetsSizeBalancedTreeMethods

```

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    {
11        [MethodImpl(MethodImplOptions.AggressiveInlining)]
12        public UInt32InternalLinksTargetsSizeBalancedTreeMethods(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(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 }

```

## 1.62 ./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,
    ↳ IndexTreeType.Default, useLinkedList: true) { }
27
28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 public UInt32SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
    ↳ this(dataMemory, indexMemory, memoryReservationStep, constants,
    ↳ IndexTreeType.Default, useLinkedList: true) { }
30
31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 public UInt32SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants,
    ↳ IndexTreeType indexTreeType, bool useLinkedList) : base(dataMemory, indexMemory,
    ↳ memoryReservationStep, constants, useLinkedList)
33 {
34     if (indexTreeType == IndexTreeType.SizeBalancedTree)
35     {
36         _createInternalSourceTreeMethods = () => new
            ↳ UInt32InternalLinksSourcesSizeBalancedTreeMethods(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
37         _createExternalSourceTreeMethods = () => new
            ↳ UInt32ExternalLinksSourcesSizeBalancedTreeMethods(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
38         _createInternalTargetTreeMethods = () => new
            ↳ UInt32InternalLinksTargetsSizeBalancedTreeMethods(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
39         _createExternalTargetTreeMethods = () => new
            ↳ UInt32ExternalLinksTargetsSizeBalancedTreeMethods(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
40     }
41     else
42     {
43         _createInternalSourceTreeMethods = () => new
            ↳ UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
44         _createExternalSourceTreeMethods = () => new
            ↳ UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
45         _createInternalTargetTreeMethods = () => new
            ↳ UInt32InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
46         _createExternalTargetTreeMethods = () => new
            ↳ UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
47     }
48     Init(dataMemory, indexMemory);
49 }
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override void SetPointers(IResizableDirectMemory dataMemory,
    ↳ IResizableDirectMemory indexMemory)
53 {
54     _linksDataParts = (RawLinkDataPart<TLink>*)dataMemory.Pointer;
55     _linksIndexParts = (RawLinkIndexPart<TLink>*)indexMemory.Pointer;
56     _header = (LinksHeader<TLink>*)indexMemory.Pointer;
57     if (_useLinkedList)
58     {
59         InternalSourcesListMethods = new
            ↳ UInt32InternalLinksSourcesLinkedListMethods(Constants, _linksDataParts,
            ↳ _linksIndexParts);
60     }
61     else
62     {
63         InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
64     }

```

```

65     ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
66     InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
67     ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
68     UnusedLinksListMethods = new UInt32UnusedLinksListMethods(_linksDataParts, _header);
69 }
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 protected override void ResetPointers()
73 {
74     base.ResetPointers();
75     _linksDataParts = null;
76     _linksIndexParts = null;
77     _header = null;
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
85     => ref _linksDataParts[linkIndex];
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
89     linkIndex) => ref _linksIndexParts[linkIndex];
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 protected override bool AreEqual(TLink first, TLink second) => first == second;
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 protected override bool LessThan(TLink first, TLink second) => first < second;
96
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 protected override bool GreaterThan(TLink first, TLink second) => first > second;
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 protected override bool GreaterOrEqualThan(TLink first, TLink second) => first >= second;
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 protected override TLink GetZero() => 0U;
108
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 protected override TLink GetOne() => 1U;
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 protected override long ConvertToInt64(TLink value) => value;
114
115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 protected override TLink ConvertToAddress(long value) => (TLink)value;
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 protected override TLink Add(TLink first, TLink second) => first + second;
120
121 [MethodImpl(MethodImplOptions.AggressiveInlining)]
122 protected override TLink Subtract(TLink first, TLink second) => first - second;
123
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 protected override TLink Increment(TLink link) => ++link;
126
127 [MethodImpl(MethodImplOptions.AggressiveInlining)]
128 protected override TLink Decrement(TLink link) => --link;
129 }

```

### 1.63 ./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     [MethodImpl(MethodImplOptions.AggressiveInlining)]
14     public UInt32UnusedLinksListMethods(RawLinkDataPart<TLink>* links, LinksHeader<TLink>*
15     ↪ header)
16         : base((byte*)links, (byte*)header)
17     {
18         _links = links;
19         _header = header;
20     }
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
24     ↪ ref _links[link];
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
28 }

```

## 1.64 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksRecursionlessSizeBalancedTree

```

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 UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase :
10     ↪ ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<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
18         ↪ UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
19         ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
20         ↪ linksIndexParts, LinksHeader<TLink>* header)
21             : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
22         {
23             LinksDataParts = linksDataParts;
24             LinksIndexParts = linksIndexParts;
25             Header = header;
26         }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override ulong GetZero() => 0UL;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override bool EqualToZero(ulong value) => value == 0UL;
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected override bool AreEqual(ulong first, ulong second) => first == second;
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected override bool GreaterThanZero(ulong value) => value > 0UL;
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override bool GreaterThan(ulong first, ulong second) => first > second;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
48         ↪ always true for ulong
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
52         ↪ always >= 0 for ulong
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
59         ↪ for ulong

```

```

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 LinksHeader<TLink> GetHeaderReference() => ref *Header;
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
74     ↪ ref LinksDataParts[link];
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
78     ↪ ref LinksIndexParts[link];
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 protected override bool FirstIsToLeftOfSecond(TLink first, TLink second)
82 {
83     ref var firstLink = ref LinksDataParts[first];
84     ref var secondLink = ref LinksDataParts[second];
85     return FirstIsToLeftOfSecond(firstLink.Source, firstLink.Target,
86         ↪ secondLink.Source, secondLink.Target);
87 }
88
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
91 {
92     ref var firstLink = ref LinksDataParts[first];
93     ref var secondLink = ref LinksDataParts[second];
94     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
95         ↪ secondLink.Source, secondLink.Target);
96 }
97 }
98 }

```

## 1.65 ./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             LinksDataParts = linksDataParts;
23             LinksIndexParts = linksIndexParts;
24             Header = header;
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override ulong GetZero() => 0UL;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override bool EqualToZero(ulong value) => value == 0UL;
32
33         [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
44     ↪ always true for ulong
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
48     ↪ always >= 0 for ulong
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
55     ↪ for ulong
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override bool LessThan(ulong first, ulong second) => first < second;
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected override ulong Increment(ulong value) => ++value;
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override ulong Decrement(ulong value) => --value;
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override ulong Add(ulong first, ulong second) => first + second;
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override ulong Subtract(ulong first, ulong 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 }

```

## 1.66 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSourcesRecursionlessSizeBalance

```

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 UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
9         ↳ UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
10    {
11        [MethodImpl(MethodImplOptions.AggressiveInlining)]
12        public
13            ↳ UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
14            ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
15            ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
16            ↳ linksIndexParts, header) { }
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        protected override ref TLink GetLeftReference(TLink node) => ref
20            ↳ LinksIndexParts[node].LeftAsSource;
21
22        [MethodImpl(MethodImplOptions.AggressiveInlining)]
23        protected override ref TLink GetRightReference(TLink node) => ref
24            ↳ LinksIndexParts[node].RightAsSource;
25
26        [MethodImpl(MethodImplOptions.AggressiveInlining)]
27        protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
28
29        [MethodImpl(MethodImplOptions.AggressiveInlining)]
30        protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
31
32        [MethodImpl(MethodImplOptions.AggressiveInlining)]
33        protected override void SetLeft(TLink node, TLink left) =>
34            ↳ LinksIndexParts[node].LeftAsSource = left;
35
36        [MethodImpl(MethodImplOptions.AggressiveInlining)]
37        protected override void SetRight(TLink node, TLink right) =>
38            ↳ LinksIndexParts[node].RightAsSource = right;
39
40        [MethodImpl(MethodImplOptions.AggressiveInlining)]
41        protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
42
43        [MethodImpl(MethodImplOptions.AggressiveInlining)]
44        protected override void SetSize(TLink node, TLink size) =>
45            ↳ LinksIndexParts[node].SizeAsSource = size;
46
47        [MethodImpl(MethodImplOptions.AggressiveInlining)]
48        protected override TLink GetTreeRoot() => Header->RootAsSource;
49
50        [MethodImpl(MethodImplOptions.AggressiveInlining)]
51        protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
52
53        [MethodImpl(MethodImplOptions.AggressiveInlining)]
54        protected override bool FirstIsToLeftOfSecond(TLink firstSource, TLink firstTarget,
55            ↳ TLink secondSource, TLink secondTarget)
56            => firstSource < secondSource || firstSource == secondSource && firstTarget <
57            ↳ secondTarget;
58
59        [MethodImpl(MethodImplOptions.AggressiveInlining)]
60        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
61            ↳ TLink secondSource, TLink secondTarget)
62            => firstSource > secondSource || firstSource == secondSource && firstTarget >
63            ↳ secondTarget;
64
65        [MethodImpl(MethodImplOptions.AggressiveInlining)]
66        protected override void ClearNode(TLink node)
67        {
68            ref var link = ref LinksIndexParts[node];
69            link.LeftAsSource = Zero;
70            link.RightAsSource = Zero;
71            link.SizeAsSource = Zero;
72        }
73    }
74 }

```

## 1.67 ./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

```

```

9 {
10     [MethodImpl(MethodImplOptions.AggressiveInlining)]
11     public UInt64ExternalLinksSourcesSizeBalancedTreeMethods(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].LeftAsSource;
15
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     protected override ref TLink GetRightReference(TLink node) => ref
        ↳ LinksIndexParts[node].RightAsSource;
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     protected override void SetLeft(TLink node, TLink left) =>
        ↳ LinksIndexParts[node].LeftAsSource = left;
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     protected override void SetRight(TLink node, TLink right) =>
        ↳ LinksIndexParts[node].RightAsSource = right;
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected override void SetSize(TLink node, TLink size) =>
        ↳ LinksIndexParts[node].SizeAsSource = size;
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override TLink GetTreeRoot() => Header->RootAsSource;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↳ TLink secondSource, TLink secondTarget)
45         => firstSource < secondSource || firstSource == secondSource && firstTarget <
        ↳ secondTarget;
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↳ TLink secondSource, TLink secondTarget)
49         => firstSource > secondSource || firstSource == secondSource && firstTarget >
        ↳ secondTarget;
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override void ClearNode(TLink node)
53     {
54         ref var link = ref LinksIndexParts[node];
55         link.LeftAsSource = Zero;
56         link.RightAsSource = Zero;
57         link.SizeAsSource = Zero;
58     }
59 }
60 }

```

## 1.68 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods

```

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 UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
        ↳ UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

11     public
12         ↳ UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(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
74 }

```

## 1.69 ./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  }

```



```

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

```

## 1.70 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksRecursionlessSizeBalancedTreeM

```

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 UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase :
10     ↪ InternalLinksRecursionlessSizeBalancedTreeMethodsBase<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
18         ↪ UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
19         ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
20         ↪ linksIndexParts, LinksHeader<TLink>* header)

```

```

17         : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
18     {
19         LinksDataParts = linksDataParts;
20         LinksIndexParts = linksIndexParts;
21         Header = header;
22     }
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     protected override ulong GetZero() => OUL;
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     protected override bool EqualToZero(ulong value) => value == OUL;
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 > OUL;
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
44     ↪ always true for ulong
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override bool LessOrEqualThanZero(ulong value) => value == OUL; // value is
48     ↪ always >= 0 for ulong
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
55     ↪ for ulong
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override bool LessThan(ulong first, ulong second) => first < second;
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected override ulong Increment(ulong value) => ++value;
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override ulong Decrement(ulong value) => --value;
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override ulong Add(ulong first, ulong second) => first + second;
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override ulong Subtract(ulong first, ulong second) => first - second;
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
74     ↪ ref LinksDataParts[link];
75
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
78     ↪ ref LinksIndexParts[link];
79
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
82     ↪ GetKeyPartValue(first) < GetKeyPartValue(second);
83
84     [MethodImpl(MethodImplOptions.AggressiveInlining)]
85     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
86     ↪ GetKeyPartValue(first) > GetKeyPartValue(second);
87
88     }
89 }

```

1.71 ./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() => 0UL;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override bool EqualToZero(ulong value) => value == 0UL;
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 > 0UL;
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 == 0UL; // 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         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             ↳ GetKeyPartValue(first) < GetKeyPartValue(second);
86     }
87 }

```

```

78     [MethodImpl(MethodImplOptions.AggressiveInlining)]
79     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
    ↪     GetKeyPartValue(first) > GetKeyPartValue(second);
80 }
81 }

```

## 1.72 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesLinkedListMethods.cs

```

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.Generic
7  {
8      public unsafe class UInt64InternalLinksSourcesLinkedListMethods :
9      ↪     InternalLinksSourcesLinkedListMethods<TLink>
10     {
11         private readonly RawLinkDataPart<TLink>* _linksDataParts;
12         private readonly RawLinkIndexPart<TLink>* _linksIndexParts;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public UInt64InternalLinksSourcesLinkedListMethods(LinksConstants<TLink> constants,
16             ↪     RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>* linksIndexParts)
17             : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts)
18         {
19             _linksDataParts = linksDataParts;
20             _linksIndexParts = linksIndexParts;
21
22             [MethodImpl(MethodImplOptions.AggressiveInlining)]
23             protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
24                 ↪     ref _linksDataParts[link];
25
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
28                 ↪     ref _linksIndexParts[link];
29         }
30     }

```

## 1.73 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesRecursionlessSizeBalance

```

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 UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods :
9      ↪     UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public
13             ↪     UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
14             ↪     constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
15             ↪     linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
16             ↪     linksIndexParts, header) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref TLink GetLeftReference(TLink node) => ref
20             ↪     LinksIndexParts[node].LeftAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ref TLink GetRightReference(TLink node) => ref
24             ↪     LinksIndexParts[node].RightAsSource;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetLeft(TLink node, TLink left) =>
34             ↪     LinksIndexParts[node].LeftAsSource = left;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override void SetRight(TLink node, TLink right) =>
38             ↪     LinksIndexParts[node].RightAsSource = right;
39     }

```

```

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

```

## 1.74 ./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 }

```

```

40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Target;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override void ClearNode(TLink node)
48     {
49         ref var link = ref LinksIndexParts[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 }

```

## 1.75 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksTargetsRecursionlessSizeBalance

```

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 UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods :
9          ↪ UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public
13         ↪ UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
14         ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
15         ↪ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
16         ↪ linksIndexParts, header) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetLeftReference(ulong node) => ref
20         ↪ LinksIndexParts[node].LeftAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ref ulong GetRightReference(ulong node) => ref
24         ↪ LinksIndexParts[node].RightAsTarget;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsTarget;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsTarget;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetLeft(TLink node, TLink left) =>
34         ↪ LinksIndexParts[node].LeftAsTarget = left;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override void SetRight(TLink node, TLink right) =>
38         ↪ LinksIndexParts[node].RightAsTarget = right;
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override void SetSize(TLink node, TLink size) =>
45         ↪ LinksIndexParts[node].SizeAsTarget = size;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override TLink GetTreeRoot(TLink node) => LinksIndexParts[node].RootAsTarget;
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Source;
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override void ClearNode(TLink node)
58         {

```

```

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) =>
56         ↪ SearchCore(GetTreeRoot(target), source);
57 }

```

## 1.76 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksTargetsSizeBalancedTreeMethod

```

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 }

```

## 1.77 ./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             ↪ IndexTreeType.Default, useLinkedList: true) { }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
34             ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
35             ↪ this(dataMemory, indexMemory, memoryReservationStep, constants,
36             ↪ IndexTreeType.Default, useLinkedList: true) { }
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
40             ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants,
41             ↪ IndexTreeType indexTreeType, bool useLinkedList) : base(dataMemory, indexMemory,
42             ↪ memoryReservationStep, constants, useLinkedList)
43         {
44             if (indexTreeType == IndexTreeType.SizeBalancedTree)
45             {
46                 _createInternalSourceTreeMethods = () => new
47                     ↪ UInt64InternalLinksSourcesSizeBalancedTreeMethods(Constants,
48                     ↪ _linksDataParts, _linksIndexParts, _header);
49                 _createExternalSourceTreeMethods = () => new
50                     ↪ UInt64ExternalLinksSourcesSizeBalancedTreeMethods(Constants,
51                     ↪ _linksDataParts, _linksIndexParts, _header);
52                 _createInternalTargetTreeMethods = () => new
53                     ↪ UInt64InternalLinksTargetsSizeBalancedTreeMethods(Constants,
54                     ↪ _linksDataParts, _linksIndexParts, _header);
55                 _createExternalTargetTreeMethods = () => new
56                     ↪ UInt64ExternalLinksTargetsSizeBalancedTreeMethods(Constants,
57                     ↪ _linksDataParts, _linksIndexParts, _header);
58             }
59             else
60             {
61                 _createInternalSourceTreeMethods = () => new
62                     ↪ UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
63                     ↪ _linksDataParts, _linksIndexParts, _header);
64                 _createExternalSourceTreeMethods = () => new
65                     ↪ UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
66                     ↪ _linksDataParts, _linksIndexParts, _header);
67                 _createInternalTargetTreeMethods = () => new
68                     ↪ UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
69                     ↪ _linksDataParts, _linksIndexParts, _header);
70                 _createExternalTargetTreeMethods = () => new
71                     ↪ UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
72                     ↪ _linksDataParts, _linksIndexParts, _header);
73             }
74             Init(dataMemory, indexMemory);
75         }
76     }
77
78     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

52     protected override void SetPointers(IResizableDirectMemory dataMemory,
    ↪ IResizableDirectMemory indexMemory)
53     {
54         _linksDataParts = (RawLinkDataPart<TLink>*)dataMemory.Pointer;
55         _linksIndexParts = (RawLinkIndexPart<TLink>*)indexMemory.Pointer;
56         _header = (LinksHeader<TLink>*)indexMemory.Pointer;
57         if (_useLinkedList)
58         {
59             InternalSourcesListMethods = new
    ↪ UInt64InternalLinksSourcesLinkedListMethods(Constants, _linksDataParts,
    ↪ _linksIndexParts);
60         }
61         else
62         {
63             InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
64         }
65         ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
66         InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
67         ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
68         UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_linksDataParts, _header);
69     }
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     protected override void ResetPointers()
73     {
74         base.ResetPointers();
75         _linksDataParts = null;
76         _linksIndexParts = null;
77         _header = null;
78     }
79
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
82
83     [MethodImpl(MethodImplOptions.AggressiveInlining)]
84     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
    ↪ => ref _linksDataParts[linkIndex];
85
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
    ↪ linkIndex) => ref _linksIndexParts[linkIndex];
88
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     protected override bool AreEqual(ulong first, ulong second) => first == second;
91
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     protected override bool LessThan(ulong first, ulong second) => first < second;
94
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
97
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     protected override bool GreaterThan(ulong first, ulong second) => first > second;
100
101     [MethodImpl(MethodImplOptions.AggressiveInlining)]
102     protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
103
104     [MethodImpl(MethodImplOptions.AggressiveInlining)]
105     protected override ulong GetZero() => 0UL;
106
107     [MethodImpl(MethodImplOptions.AggressiveInlining)]
108     protected override ulong GetOne() => 1UL;
109
110     [MethodImpl(MethodImplOptions.AggressiveInlining)]
111     protected override long ConvertToInt64(ulong value) => (long)value;
112
113     [MethodImpl(MethodImplOptions.AggressiveInlining)]
114     protected override ulong ConvertToAddress(long value) => (ulong)value;
115
116     [MethodImpl(MethodImplOptions.AggressiveInlining)]
117     protected override ulong Add(ulong first, ulong second) => first + second;
118
119     [MethodImpl(MethodImplOptions.AggressiveInlining)]
120     protected override ulong Subtract(ulong first, ulong second) => first - second;
121
122     [MethodImpl(MethodImplOptions.AggressiveInlining)]
123     protected override ulong Increment(ulong link) => ++link;
124
125     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

126         protected override ulong Decrement(ulong link) => --link;
127     }
128 }

```

## 1.78 ./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 }

```

## 1.79 ./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 }

```

```

38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 protected abstract TLink GetBasePartValue(TLink link);
40
41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
43     ↪ rootSource, TLink rootTarget);
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
47     ↪ rootSource, TLink rootTarget);
48
49 [MethodImpl(MethodImplOptions.AggressiveInlining)]
50 protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
51     ↪ AsRef<LinksHeader<TLink>>(Header);
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
55     ↪ AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
56     ↪ _addressToInt64Converter.Convert(link)));
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
60 {
61     ref var link = ref GetLinkReference(linkIndex);
62     return new Link<TLink>(linkIndex, link.Source, link.Target);
63 }
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
67 {
68     ref var firstLink = ref GetLinkReference(first);
69     ref var secondLink = ref GetLinkReference(second);
70     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
71     ↪ secondLink.Source, secondLink.Target);
72 }
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
76 {
77     ref var firstLink = ref GetLinkReference(first);
78     ref var secondLink = ref GetLinkReference(second);
79     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
80     ↪ secondLink.Source, secondLink.Target);
81 }
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 protected virtual TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5,
85     ↪ -5);
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 protected virtual void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
89     ↪ Bit<TLink>.PartialWrite(storedValue, size, 5, -5);
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 protected virtual bool GetLeftIsChildValue(TLink value)
93 {
94     unchecked
95     {
96         return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 4, 1));
97         //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
98     }
99 }
100
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 protected virtual void SetLeftIsChildValue(ref TLink storedValue, bool value)
103 {
104     unchecked
105     {
106         var previousValue = storedValue;
107         var modified = Bit<TLink>.PartialWrite(previousValue,
108     ↪ _boolToAddressConverter.Convert(value), 4, 1);
109         storedValue = modified;
110     }
111 }
112
113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
114 protected virtual bool GetRightIsChildValue(TLink value)

```

```

106 {
107     unchecked
108     {
109         return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 3, 1));
110         //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
111     }
112 }
113
114 [MethodImpl(MethodImplOptions.AggressiveInlining)]
115 protected virtual void SetRightIsChildValue(ref TLink storedValue, bool value)
116 {
117     unchecked
118     {
119         var previousValue = storedValue;
120         var modified = Bit<TLink>.PartialWrite(previousValue,
121             ↪ _boolToAddressConverter.Convert(value), 3, 1);
122         storedValue = modified;
123     }
124 }
125
126 [MethodImpl(MethodImplOptions.AggressiveInlining)]
127 protected bool IsChild(TLink parent, TLink possibleChild)
128 {
129     var parentSize = GetSize(parent);
130     var childSize = GetSizeOrZero(possibleChild);
131     return GreaterThanZero(childSize) && LessOrEqualThan(childSize, parentSize);
132 }
133
134 [MethodImpl(MethodImplOptions.AggressiveInlining)]
135 protected virtual sbyte GetBalanceValue(TLink storedValue)
136 {
137     unchecked
138     {
139         var value = _addressToInt32Converter.Convert(Bit<TLink>.PartialRead(storedValue,
140             ↪ 0, 3));
141         value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
142             ↪ end of sbyte
143         return (sbyte)value;
144     }
145 }
146
147 [MethodImpl(MethodImplOptions.AggressiveInlining)]
148 protected virtual void SetBalanceValue(ref TLink storedValue, sbyte value)
149 {
150     unchecked
151     {
152         var packagedValue = _int32ToAddressConverter.Convert((byte)value >> 5 & 4 |
153             ↪ value & 3);
154         var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
155         storedValue = modified;
156     }
157 }
158
159 public TLink this[TLink index]
160 {
161     [MethodImpl(MethodImplOptions.AggressiveInlining)]
162     get
163     {
164         var root = GetTreeRoot();
165         if (GreaterOrEqualThan(index, GetSize(root)))
166         {
167             return Zero;
168         }
169         while (!EqualToZero(root))
170         {
171             var left = GetLeftOrDefault(root);
172             var leftSize = GetSizeOrZero(left);
173             if (LessThan(index, leftSize))
174             {
175                 root = left;
176                 continue;
177             }
178             if (AreEqual(index, leftSize))
179             {
180                 return root;
181             }
182             root = GetRightOrDefault(root);
183             index = Subtract(index, Increment(leftSize));
184         }
185     }
186 }

```

```

181         return Zero; // TODO: Impossible situation exception (only if tree structure
182         ↪ broken)
183     }
184 }
185
186 /// <summary>
187 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
188 ↪ (концом).
189 /// </summary>
190 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
191 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
192 /// <returns>Индекс искомой связи.</returns>
193 [MethodImpl(MethodImplOptions.AggressiveInlining)]
194 public TLink Search(TLink source, TLink target)
195 {
196     var root = GetTreeRoot();
197     while (!EqualToZero(root))
198     {
199         ref var rootLink = ref GetLinkReference(root);
200         var rootSource = rootLink.Source;
201         var rootTarget = rootLink.Target;
202         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
203             ↪ node.Key < root.Key
204         {
205             root = GetLeftOrDefault(root);
206         }
207         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
208             ↪ node.Key > root.Key
209         {
210             root = GetRightOrDefault(root);
211         }
212         else // node.Key == root.Key
213         {
214             return root;
215         }
216     }
217     return Zero;
218 }
219
220 // TODO: Return indices range instead of references count
221 [MethodImpl(MethodImplOptions.AggressiveInlining)]
222 public TLink CountUsages(TLink link)
223 {
224     var root = GetTreeRoot();
225     var total = GetSize(root);
226     var totalRightIgnore = Zero;
227     while (!EqualToZero(root))
228     {
229         var @base = GetBasePartValue(root);
230         if (LessOrEqualThan(@base, link))
231         {
232             root = GetRightOrDefault(root);
233         }
234         else
235         {
236             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
237             root = GetLeftOrDefault(root);
238         }
239     }
240     root = GetTreeRoot();
241     var totalLeftIgnore = Zero;
242     while (!EqualToZero(root))
243     {
244         var @base = GetBasePartValue(root);
245         if (GreaterOrEqualThan(@base, link))
246         {
247             root = GetLeftOrDefault(root);
248         }
249         else
250         {
251             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
252             root = GetRightOrDefault(root);
253         }
254     }
255     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
256 }

```

```

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

```

## 1.80 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksRecursionlessSizeBalancedTreeMethodsBase

```

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 LinksRecursionlessSizeBalancedTreeMethodsBase<TLink> :
14         ↳ RecursionlessSizeBalancedTreeMethods<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;

```

```

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

```

```

91         if (AreEqual(index, leftSize))
92         {
93             return root;
94         }
95         root = GetRightOrDefault(root);
96         index = Subtract(index, Increment(leftSize));
97     }
98     return Zero; // TODO: Impossible situation exception (only if tree structure
99     ↪ broken)
100 }
101
102 /// <summary>
103 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
104 ↪ (концом).
105 /// </summary>
106 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
107 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
108 /// <returns>Индекс искомой связи.</returns>
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 public TLink Search(TLink source, TLink target)
111 {
112     var root = GetTreeRoot();
113     while (!EqualToZero(root))
114     {
115         ref var rootLink = ref GetLinkReference(root);
116         var rootSource = rootLink.Source;
117         var rootTarget = rootLink.Target;
118         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
119             ↪ node.Key < root.Key
120         {
121             root = GetLeftOrDefault(root);
122         }
123         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
124             ↪ node.Key > root.Key
125         {
126             root = GetRightOrDefault(root);
127         }
128         else // node.Key == root.Key
129         {
130             return root;
131         }
132     }
133     return Zero;
134 }
135
136 // TODO: Return indices range instead of references count
137 [MethodImpl(MethodImplOptions.AggressiveInlining)]
138 public TLink CountUsages(TLink link)
139 {
140     var root = GetTreeRoot();
141     var total = GetSize(root);
142     var totalRightIgnore = Zero;
143     while (!EqualToZero(root))
144     {
145         var @base = GetBasePartValue(root);
146         if (LessOrEqualThan(@base, link))
147         {
148             root = GetRightOrDefault(root);
149         }
150         else
151         {
152             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
153             root = GetLeftOrDefault(root);
154         }
155     }
156     root = GetTreeRoot();
157     var totalLeftIgnore = Zero;
158     while (!EqualToZero(root))
159     {
160         var @base = GetBasePartValue(root);
161         if (GreaterOrEqualThan(@base, link))
162         {
163             root = GetLeftOrDefault(root);
164         }
165         else
166         {
167             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
168         }
169     }
170 }

```



```

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

```

## 1.81 ./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> :
        ↳ SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
14     {
15         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            ↳ UncheckedConverter<TLink, long>.Default;
16
17         protected readonly TLink Break;
18         protected readonly TLink Continue;
19         protected readonly byte* Links;
20         protected readonly byte* Header;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
            ↳ byte* header)
24         {
25             Links = links;
26             Header = header;
27             Break = constants.Break;
28             Continue = constants.Continue;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected abstract TLink GetTreeRoot();
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected abstract TLink GetBasePartValue(TLink link);
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
            ↳ rootSource, TLink rootTarget);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
            ↳ rootSource, TLink rootTarget);
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
            ↳ AsRef<LinksHeader<TLink>>(Header);
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
            ↳ AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
            ↳ _addressToInt64Converter.Convert(link)));
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
51         {
52             ref var link = ref GetLinkReference(linkIndex);
53             return new Link<TLink>(linkIndex, link.Source, link.Target);
54         }
55
56         [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
99     ↪ broken)
100 }
101
102 /// <summary>
103 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
104 ↪ (концом).
105 /// </summary>
106 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
107 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
108 /// <returns>Индекс искомой связи.</returns>
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 public TLink Search(TLink source, TLink target)
111 {
112     var root = GetTreeRoot();
113     while (!EqualToZero(root))
114     {
115         ref var rootLink = ref GetLinkReference(root);
116         var rootSource = rootLink.Source;
117         var rootTarget = rootLink.Target;
118         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
119             ↪ node.Key < root.Key
120         {
121             root = GetLeftOrDefault(root);
122         }
123         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
124             ↪ node.Key > root.Key
125         {
126             root = GetRightOrDefault(root);
127         }
128         else // node.Key == root.Key
129         {
130             return root;
131         }
132     }
133     return Zero;
134 }
135
136 // TODO: Return indices range instead of references count
137 [MethodImpl(MethodImplOptions.AggressiveInlining)]
138 public TLink CountUsages(TLink link)
139 {
140     var root = GetTreeRoot();
141     var total = GetSize(root);
142     var totalRightIgnore = Zero;
143     while (!EqualToZero(root))
144     {
145         var @base = GetBasePartValue(root);
146         if (LessOrEqualThan(@base, link))
147         {
148             root = GetRightOrDefault(root);
149         }
150         else
151         {
152             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
153             root = GetLeftOrDefault(root);
154         }
155     }
156     root = GetTreeRoot();
157     var totalLeftIgnore = Zero;

```

```

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

```

```

1.82 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesAvlBalancedTreeMethods.cs
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.United.Generic
6  {
7      public unsafe class LinksSourcesAvlBalancedTreeMethods<TLink> :
8          ↳ LinksAvlBalancedTreeMethodsBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksSourcesAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
12             ↳ byte* header) : base(constants, links, header) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref TLink GetLeftReference(TLink node) => ref
16             ↳ GetLinkReference(node).LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref TLink GetRightReference(TLink node) => ref
20             ↳ GetLinkReference(node).RightAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(TLink node, TLink left) =>
30             ↳ GetLinkReference(node).LeftAsSource = left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(TLink node, TLink right) =>
34             ↳ GetLinkReference(node).RightAsSource = right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override TLink GetSize(TLink node) =>
38             ↳ GetSizeValue(GetLinkReference(node).SizeAsSource);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
42             ↳ GetLinkReference(node).SizeAsSource, size);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override bool GetLeftIsChild(TLink node) =>
46             ↳ GetLeftIsChildValue(GetLinkReference(node).SizeAsSource);
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected override void SetLeftIsChild(TLink node, bool value) =>
50             ↳ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override bool GetRightIsChild(TLink node) =>
54             ↳ GetRightIsChildValue(GetLinkReference(node).SizeAsSource);
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override void SetRightIsChild(TLink node, bool value) =>
58             ↳ SetRightIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         protected override sbyte GetBalance(TLink node) =>
62             ↳ GetBalanceValue(GetLinkReference(node).SizeAsSource);
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
66             ↳ GetLinkReference(node).SizeAsSource, value);
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
70
71         [MethodImpl(MethodImplOptions.AggressiveInlining)]
72         protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
73
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]
75         protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
76             ↳ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
77             ↳ (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
78     }
79 }

```

```

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

```

1.83 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesRecursionlessSizeBalancedTreeMethods

```

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 LinksSourcesRecursionlessSizeBalancedTreeMethods<TLink> :
8     ↪ LinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink> constants,
12         ↪ byte* links, byte* header) : base(constants, links, header) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref TLink GetLeftReference(TLink node) => ref
16         ↪ GetLinkReference(node).LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref TLink GetRightReference(TLink node) => ref
20         ↪ GetLinkReference(node).RightAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(TLink node, TLink left) =>
30         ↪ GetLinkReference(node).LeftAsSource = left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(TLink node, TLink right) =>
34         ↪ GetLinkReference(node).RightAsSource = right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsSource;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(TLink node, TLink size) =>
41         ↪ GetLinkReference(node).SizeAsSource = size;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
51         ↪ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
            ↪ (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
55         ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
            ↪ (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         protected override void ClearNode(TLink node)
59         {

```

```

51         ref var link = ref GetLinkReference(node);
52         link.LeftAsSource = Zero;
53         link.RightAsSource = Zero;
54         link.SizeAsSource = Zero;
55     }
56 }
57 }

```

#### 1.84 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesSizeBalancedTreeMethods.cs

```

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

```

```

1.85 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsAvlBalancedTreeMethods.cs
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.United.Generic
6  {
7      public unsafe class LinksTargetsAvlBalancedTreeMethods<TLink> :
8          ↳ LinksAvlBalancedTreeMethodsBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksTargetsAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
12             ↳ byte* header) : base(constants, links, header) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref TLink GetLeftReference(TLink node) => ref
16             ↳ GetLinkReference(node).LeftAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref TLink GetRightReference(TLink node) => ref
20             ↳ GetLinkReference(node).RightAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(TLink node, TLink left) =>
30             ↳ GetLinkReference(node).LeftAsTarget = left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(TLink node, TLink right) =>
34             ↳ GetLinkReference(node).RightAsTarget = right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override TLink GetSize(TLink node) =>
38             ↳ GetSizeValue(GetLinkReference(node).SizeAsTarget);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
42             ↳ GetLinkReference(node).SizeAsTarget, size);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override bool GetLeftIsChild(TLink node) =>
46             ↳ GetLeftIsChildValue(GetLinkReference(node).SizeAsTarget);
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected override void SetLeftIsChild(TLink node, bool value) =>
50             ↳ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override bool GetRightIsChild(TLink node) =>
54             ↳ GetRightIsChildValue(GetLinkReference(node).SizeAsTarget);
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override void SetRightIsChild(TLink node, bool value) =>
58             ↳ SetRightIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         protected override sbyte GetBalance(TLink node) =>
62             ↳ GetBalanceValue(GetLinkReference(node).SizeAsTarget);
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
66             ↳ GetLinkReference(node).SizeAsTarget, value);
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;
70
71         [MethodImpl(MethodImplOptions.AggressiveInlining)]
72         protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
73
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]
75         protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
76             ↳ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
77             ↳ (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
78     }
79 }

```



```

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

```

## 1.86 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsRecursionlessSizeBalancedTreeMethods

```

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

```

```

51         ref var link = ref GetLinkReference(node);
52         link.LeftAsTarget = Zero;
53         link.RightAsTarget = Zero;
54         link.SizeAsTarget = Zero;
55     }
56 }
57 }

```

## 1.87 ./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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(TLink node, TLink right) =>
34             ↳ GetLinkReference(node).RightAsTarget = right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsTarget;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(TLink node, TLink size) =>
41             ↳ GetLinkReference(node).SizeAsTarget = size;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
51             ↳ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
52             ↳ (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
56             ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
57             ↳ (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override void ClearNode(TLink node)
61         {
62             ref var link = ref GetLinkReference(node);
63             link.LeftAsTarget = Zero;
64             link.RightAsTarget = Zero;
65             link.SizeAsTarget = Zero;
66         }
67     }
68 }

```

## 1.88 ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinks.cs

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using static System.Runtime.CompilerServices.Unsafe;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Memory.United.Generic
10 {
11     public unsafe class UnitedMemoryLinks<TLink> : UnitedMemoryLinksBase<TLink>
12     {
13         private readonly Func<ILinksTreeMethods<TLink>> _createSourceTreeMethods;
14         private readonly Func<ILinksTreeMethods<TLink>> _createTargetTreeMethods;
15         private byte* _header;
16         private byte* _links;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
20
21         /// <summary>
22         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
23         /// → минимальным шагом расширения базы данных.
24         /// </summary>
25         /// <param name="address">Полный путь к файлу базы данных.</param>
26         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
27         /// → байтах.</param>
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
30         {
31             ↪ FileMappedResizableDirectMemory(address, memoryReservationStep),
32             ↪ memoryReservationStep
33         }) { }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
37         {
38             ↪ DefaultLinksSizeStep
39         }) { }
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep) :
43         {
44             ↪ this(memory, memoryReservationStep, Default<LinksConstants<TLink>>.Instance,
45             ↪ IndexTreeType.Default)
46         } { }
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep,
50         {
51             ↪ LinksConstants<TLink> constants, IndexTreeType indexTreeType) : base(memory,
52             ↪ memoryReservationStep, constants)
53         {
54             if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
55             {
56                 _createSourceTreeMethods = () => new
57                 {
58                     ↪ LinksSourcesAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
59                 };
60                 _createTargetTreeMethods = () => new
61                 {
62                     ↪ LinksTargetsAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
63                 };
64             }
65             else
66             {
67                 _createSourceTreeMethods = () => new
68                 {
69                     ↪ LinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
70                 };
71                 _createTargetTreeMethods = () => new
72                 {
73                     ↪ LinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
74                 };
75             }
76             Init(memory, memoryReservationStep);
77         }
78
79         [MethodImpl(MethodImplOptions.AggressiveInlining)]
80         protected override void SetPointers(IResizableDirectMemory memory)
81         {
82             _links = (byte*)memory.Pointer;
83             _header = _links;
84             SourcesTreeMethods = _createSourceTreeMethods();
85             TargetsTreeMethods = _createTargetTreeMethods();
86             UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_links, _header);
87         }
88
89         [MethodImpl(MethodImplOptions.AggressiveInlining)]
90         protected override void ResetPointers()
91         {
92             base.ResetPointers();
93         }
94     }
95 }

```

```

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 }

```

## 1.89 ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinksBase.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5  using Platform.Singletons;
6  using Platform.Converters;
7  using Platform.Numbers;
8  using Platform.Memory;
9  using Platform.Data.Exceptions;
10
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Data.Doublets.Memory.United.Generic
14 {
15     public abstract class UnitedMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16     {
17         private static readonly EqualityComparer<TLink> _equalityComparer =
18             ↳ EqualityComparer<TLink>.Default;
19         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
20         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
21             ↳ UncheckedConverter<TLink, long>.Default;
22         private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
23             ↳ UncheckedConverter<long, TLink>.Default;
24
25         private static readonly TLink _zero = default;
26         private static readonly TLink _one = Arithmetic.Increment(_zero);
27
28         /// <summary>Возвращает размер одной связи в байтах.</summary>
29         /// <remarks>
30         ///     Используется только во вне класса, не рекомендуется использовать внутри.
31         ///     Так как во вне не обязательно будет доступен unsafe C#.
32         /// </remarks>
33         public static readonly long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;
34
35         public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
36
37         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
38
39         protected readonly IResizableDirectMemory _memory;
40         protected readonly long _memoryReservationStep;
41
42         protected ILinksTreeMethods<TLink> TargetsTreeMethods;
43         protected ILinksTreeMethods<TLink> SourcesTreeMethods;
44         // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
45         // ↳ нужно использовать не список а дерево, так как так можно быстрее проверить на
46         // ↳ наличие связи внутри
47         protected ILinksListMethods<TLink> UnusedLinksListMethods;
48
49         /// <summary>
50         ///     Возвращает общее число связей находящихся в хранилище.
51         /// </summary>
52         protected virtual TLink Total
53         {
54             [MethodImpl(MethodImplOptions.AggressiveInlining)]
55             get
56             {
57                 ref var header = ref GetHeaderReference();
58                 return Subtract(header.AllocatedLinks, header.FreeLinks);
59             }
60         }
61
62         public virtual LinksConstants<TLink> Constants
63         {
64             [MethodImpl(MethodImplOptions.AggressiveInlining)]
65             get;
66         }
67     }
68 }

```

```

62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
64     ↪ memoryReservationStep, LinksConstants<TLink> constants)
65 {
66     _memory = memory;
67     _memoryReservationStep = memoryReservationStep;
68     Constants = constants;
69 }
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
    ↪ memoryReservationStep) : this(memory, memoryReservationStep,
    ↪ Default<LinksConstants<TLink>>.Instance) { }
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
76 {
77     if (memory.ReservedCapacity < memoryReservationStep)
78     {
79         memory.ReservedCapacity = memoryReservationStep;
80     }
81     SetPointers(memory);
82     ref var header = ref GetHeaderReference();
83     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
84     memory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) * LinkSizeInBytes) +
    ↪ LinkHeaderSizeInBytes;
85     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
86     header.ReservedLinks = ConvertToAddress((memory.ReservedCapacity -
    ↪ LinkHeaderSizeInBytes) / LinkSizeInBytes);
87 }
88
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 public virtual TLink Count(ICollection<TLink> restrictions)
91 {
92     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
93     if (restrictions.Count == 0)
94     {
95         return Total;
96     }
97     var constants = Constants;
98     var any = constants.Any;
99     var index = restrictions[constants.IndexPart];
100     if (restrictions.Count == 1)
101     {
102         if (AreEqual(index, any))
103         {
104             return Total;
105         }
106         return Exists(index) ? GetOne() : GetZero();
107     }
108     if (restrictions.Count == 2)
109     {
110         var value = restrictions[1];
111         if (AreEqual(index, any))
112         {
113             if (AreEqual(value, any))
114             {
115                 return Total; // Any - как отсутствие ограничения
116             }
117             return Add(SourcesTreeMethods.CountUsages(value),
    ↪ TargetsTreeMethods.CountUsages(value));
118         }
119         else
120         {
121             if (!Exists(index))
122             {
123                 return GetZero();
124             }
125             if (AreEqual(value, any))
126             {
127                 return GetOne();
128             }
129             ref var storedLinkValue = ref GetLinkReference(index);
130             if (AreEqual(storedLinkValue.Source, value) ||
    ↪ AreEqual(storedLinkValue.Target, value))
131             {
132                 return GetOne();

```

```

133     }
134     return GetZero();
135 }
136 }
137 if (restrictions.Count == 3)
138 {
139     var source = restrictions[constants.SourcePart];
140     var target = restrictions[constants.TargetPart];
141     if (AreEqual(index, any))
142     {
143         if (AreEqual(source, any) && AreEqual(target, any))
144         {
145             return Total;
146         }
147         else if (AreEqual(source, any))
148         {
149             return TargetsTreeMethods.CountUsages(target);
150         }
151         else if (AreEqual(target, any))
152         {
153             return SourcesTreeMethods.CountUsages(source);
154         }
155         else //if(source != Any && target != Any)
156         {
157             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
158             var link = SourcesTreeMethods.Search(source, target);
159             return AreEqual(link, constants.Null) ? GetZero() : GetOne();
160         }
161     }
162     else
163     {
164         if (!Exists(index))
165         {
166             return GetZero();
167         }
168         if (AreEqual(source, any) && AreEqual(target, any))
169         {
170             return GetOne();
171         }
172         ref var storedLinkValue = ref GetLinkReference(index);
173         if (!AreEqual(source, any) && !AreEqual(target, any))
174         {
175             if (AreEqual(storedLinkValue.Source, source) &&
176                 ⇨ AreEqual(storedLinkValue.Target, target))
177             {
178                 return GetOne();
179             }
180             return GetZero();
181         }
182         var value = default(TLink);
183         if (AreEqual(source, any))
184         {
185             value = target;
186         }
187         if (AreEqual(target, any))
188         {
189             value = source;
190         }
191         if (AreEqual(storedLinkValue.Source, value) ||
192             ⇨ AreEqual(storedLinkValue.Target, value))
193         {
194             return GetOne();
195         }
196         return GetZero();
197     }
198 }
199 }
200 throw new NotSupportedException("Другие размеры и способы ограничений не
201 ⇨ поддерживаются.");
202 }
203 [MethodImpl(MethodImplOptions.AggressiveInlining)]
204 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
205 {
206     var constants = Constants;
207     var @break = constants.Break;
208     if (restrictions.Count == 0)
209     {

```

```

207     for (var link = GetOne(); LessOrEqualThan(link,
208 ↪ GetHeaderReference().AllocatedLinks); link = Increment(link))
209     {
210         if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
211         {
212             return @break;
213         }
214     }
215     return @break;
216 }
217 var @continue = constants.Continue;
218 var any = constants.Any;
219 var index = restrictions[constants.IndexPart];
220 if (restrictions.Count == 1)
221 {
222     if (AreEqual(index, any))
223     {
224         return Each(handler, Array.Empty<TLink>());
225     }
226     if (!Exists(index))
227     {
228         return @continue;
229     }
230     return handler(GetLinkStruct(index));
231 }
232 if (restrictions.Count == 2)
233 {
234     var value = restrictions[1];
235     if (AreEqual(index, any))
236     {
237         if (AreEqual(value, any))
238         {
239             return Each(handler, Array.Empty<TLink>());
240         }
241         if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
242         {
243             return @break;
244         }
245         return Each(handler, new Link<TLink>(index, any, value));
246     }
247     else
248     {
249         if (!Exists(index))
250         {
251             return @continue;
252         }
253         if (AreEqual(value, any))
254         {
255             return handler(GetLinkStruct(index));
256         }
257         ref var storedLinkValue = ref GetLinkReference(index);
258         if (AreEqual(storedLinkValue.Source, value) ||
259             AreEqual(storedLinkValue.Target, value))
260         {
261             return handler(GetLinkStruct(index));
262         }
263         return @continue;
264     }
265 }
266 if (restrictions.Count == 3)
267 {
268     var source = restrictions[constants.SourcePart];
269     var target = restrictions[constants.TargetPart];
270     if (AreEqual(index, any))
271     {
272         if (AreEqual(source, any) && AreEqual(target, any))
273         {
274             return Each(handler, Array.Empty<TLink>());
275         }
276         else if (AreEqual(source, any))
277         {
278             return TargetsTreeMethods.EachUsage(target, handler);
279         }
280         else if (AreEqual(target, any))
281         {
282             return SourcesTreeMethods.EachUsage(source, handler);
283         }
284         else //if(source != Any && target != Any)

```

```

284         {
285             var link = SourcesTreeMethods.Search(source, target);
286             return AreEqual(link, constants.Null) ? @continue :
                ↪ handler(GetLinkStruct(link));
287         }
288     }
289     else
290     {
291         if (!Exists(index))
292         {
293             return @continue;
294         }
295         if (AreEqual(source, any) && AreEqual(target, any))
296         {
297             return handler(GetLinkStruct(index));
298         }
299         ref var storedLinkValue = ref GetLinkReference(index);
300         if (!AreEqual(source, any) && !AreEqual(target, any))
301         {
302             if (AreEqual(storedLinkValue.Source, source) &&
303                 AreEqual(storedLinkValue.Target, target))
304             {
305                 return handler(GetLinkStruct(index));
306             }
307             return @continue;
308         }
309         var value = default(TLink);
310         if (AreEqual(source, any))
311         {
312             value = target;
313         }
314         if (AreEqual(target, any))
315         {
316             value = source;
317         }
318         if (AreEqual(storedLinkValue.Source, value) ||
319             AreEqual(storedLinkValue.Target, value))
320         {
321             return handler(GetLinkStruct(index));
322         }
323         return @continue;
324     }
325 }
326 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↪ поддерживаются.");
327 }
328
329 /// <remarks>
330 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
331 ↪ в другом месте (но не в менеджере памяти, а в логике Links)
332 /// </remarks>
333 [MethodImpl(MethodImplOptions.AggressiveInlining)]
334 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
335 {
336     var constants = Constants;
337     var @null = constants.Null;
338     var linkIndex = restrictions[constants.IndexPart];
339     ref var link = ref GetLinkReference(linkIndex);
340     ref var header = ref GetHeaderReference();
341     ref var firstAsSource = ref header.RootAsSource;
342     ref var firstAsTarget = ref header.RootAsTarget;
343     // Будет корректно работать только в том случае, если пространство выделенной связи
344     ↪ предварительно заполнено нулями
345     if (!AreEqual(link.Source, @null))
346     {
347         SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
348     }
349     if (!AreEqual(link.Target, @null))
350     {
351         TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
352     }
353     link.Source = substitution[constants.SourcePart];
354     link.Target = substitution[constants.TargetPart];
355     if (!AreEqual(link.Source, @null))
356     {
357         SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
358     }
359     if (!AreEqual(link.Target, @null))

```



```

358     {
359         TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
360     }
361     return linkIndex;
362 }
363
364 /// <remarks>
365 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
366 /// ↪ пространство
367 /// </remarks>
368 [MethodImpl(MethodImplOptions.AggressiveInlining)]
369 public virtual TLink Create(ICollection<TLink> restrictions)
370 {
371     ref var header = ref GetHeaderReference();
372     var freeLink = header.FirstFreeLink;
373     if (!AreEqual(freeLink, Constants.Null))
374     {
375         UnusedLinksListMethods.Detach(freeLink);
376     }
377     else
378     {
379         var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
380         if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
381         {
382             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
383         }
384         if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks))
385         {
386             _memory.ReservedCapacity += _memory.ReservationStep;
387             SetPointers(_memory);
388             header = ref GetHeaderReference();
389             header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
390             ↪ LinkSizeInBytes);
391         }
392         freeLink = header.AllocatedLinks = Increment(header.AllocatedLinks);
393         _memory.UsedCapacity += LinkSizeInBytes;
394     }
395     return freeLink;
396 }
397
398 [MethodImpl(MethodImplOptions.AggressiveInlining)]
399 public virtual void Delete(ICollection<TLink> restrictions)
400 {
401     ref var header = ref GetHeaderReference();
402     var link = restrictions[Constants.IndexPart];
403     if (LessThan(link, header.AllocatedLinks)
404     {
405         UnusedLinksListMethods.AttachAsFirst(link);
406     }
407     else if (AreEqual(link, header.AllocatedLinks)
408     {
409         header.AllocatedLinks = Decrement(header.AllocatedLinks);
410         _memory.UsedCapacity -= LinkSizeInBytes;
411         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
412         // ↪ пока не дойдём до первой существующей связи
413         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
414         while (GreaterThan(header.AllocatedLinks, GetZero()) &&
415         ↪ IsUnusedLink(header.AllocatedLinks))
416         {
417             UnusedLinksListMethods.Detach(header.AllocatedLinks);
418             header.AllocatedLinks = Decrement(header.AllocatedLinks);
419             _memory.UsedCapacity -= LinkSizeInBytes;
420         }
421     }
422 }
423
424 [MethodImpl(MethodImplOptions.AggressiveInlining)]
425 public IList<TLink> GetLinkStruct(TLink linkIndex)
426 {
427     ref var link = ref GetLinkReference(linkIndex);
428     return new Link<TLink>(linkIndex, link.Source, link.Target);
429 }
430
431 /// <remarks>
432 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
433 /// ↪ адрес реально поменялся
434 ///
435 /// Указатель this.links может быть в том же месте,
436 /// так как 0-я связь не используется и имеет такой же размер как Header,

```

```

432 /// поэтому header размещается в том же месте, что и 0-я связь
433 /// </remarks>
434 [MethodImpl(MethodImplOptions.AggressiveInlining)]
435 protected abstract void SetPointers(IResizableDirectMemory memory);
436
437 [MethodImpl(MethodImplOptions.AggressiveInlining)]
438 protected virtual void ResetPointers()
439 {
440     SourcesTreeMethods = null;
441     TargetsTreeMethods = null;
442     UnusedLinksListMethods = null;
443 }
444
445 [MethodImpl(MethodImplOptions.AggressiveInlining)]
446 protected abstract ref LinksHeader<TLink> GetHeaderReference();
447
448 [MethodImpl(MethodImplOptions.AggressiveInlining)]
449 protected abstract ref RawLink<TLink> GetLinkReference(TLink linkIndex);
450
451 [MethodImpl(MethodImplOptions.AggressiveInlining)]
452 protected virtual bool Exists(TLink link)
453     => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
454         && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
455         && !IsUnusedLink(link);
456
457 [MethodImpl(MethodImplOptions.AggressiveInlining)]
458 protected virtual bool IsUnusedLink(TLink linkIndex)
459 {
460     if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
461         ↪ is not needed
462     {
463         ref var link = ref GetLinkReference(linkIndex);
464         return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
465     }
466     else
467     {
468         return true;
469     }
470 }
471
472 [MethodImpl(MethodImplOptions.AggressiveInlining)]
473 protected virtual TLink GetOne() => _one;
474
475 [MethodImpl(MethodImplOptions.AggressiveInlining)]
476 protected virtual TLink GetZero() => default;
477
478 [MethodImpl(MethodImplOptions.AggressiveInlining)]
479 protected virtual bool AreEqual(TLink first, TLink second) =>
480     ↪ _equalityComparer.Equals(first, second);
481
482 [MethodImpl(MethodImplOptions.AggressiveInlining)]
483 protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
484     ↪ second) < 0;
485
486 [MethodImpl(MethodImplOptions.AggressiveInlining)]
487 protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
488     ↪ _comparer.Compare(first, second) <= 0;
489
490 [MethodImpl(MethodImplOptions.AggressiveInlining)]
491 protected virtual bool GreaterThan(TLink first, TLink second) =>
492     ↪ _comparer.Compare(first, second) > 0;
493
494 [MethodImpl(MethodImplOptions.AggressiveInlining)]
495 protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
496     ↪ _comparer.Compare(first, second) >= 0;
497
498 [MethodImpl(MethodImplOptions.AggressiveInlining)]
499 protected virtual long ConvertToInt64(TLink value) =>
500     ↪ _addressToInt64Converter.Convert(value);
501
502 [MethodImpl(MethodImplOptions.AggressiveInlining)]
503 protected virtual TLink ConvertToAddress(long value) =>
504     ↪ _int64ToAddressConverter.Convert(value);
505
506 [MethodImpl(MethodImplOptions.AggressiveInlining)]
507 protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
508     ↪ second);
509
510 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

## 1.90 ./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> :
11         ↪ AbsoluteCircularDoublyLinkedListMethods<TLink>, ILinksListMethods<TLink>
12     {
13         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
14             ↪ UncheckedConverter<TLink, long>.Default;
15
16         private readonly byte* _links;
17         private readonly byte* _header;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnusedLinksListMethods(byte* links, byte* header)
21         {
22             _links = links;
23             _header = header;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
28             ↪ AsRef<LinksHeader<TLink>>(_header);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
32             ↪ AsRef<RawLink<TLink>>(_links + (RawLink<TLink>.SizeInBytes *
33             ↪ _addressToInt64Converter.Convert(link)));
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected override TLink GetPrevious(TLink element) => GetLinkReference(element).Source;
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override TLink GetNext(TLink element) => GetLinkReference(element).Target;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override TLink GetSize() => GetHeaderReference().FreeLinks;

```

```

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

```

### 1.91 ./csharp/Platform.Data.Doublets/Memory/United/RawLink.cs

```

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

```

### 1.92 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksRecursionlessSizeBalancedTreeMethod

```

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 UInt32LinksRecursionlessSizeBalancedTreeMethodsBase :
9         ↳ LinksRecursionlessSizeBalancedTreeMethodsBase<uint>
10     {
11         protected new readonly RawLink<uint>* Links;
12         protected new readonly LinksHeader<uint>* Header;
13
14         protected UInt32LinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<uint>
15             ↳ constants, 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
41             ↳ always true for uint
42
43             [MethodImpl(MethodImplOptions.AggressiveInlining)]
44             protected override bool LessOrEqualThanZero(uint value) => value == 0U; // value is
45             ↳ always >= 0 for uint
46
47             [MethodImpl(MethodImplOptions.AggressiveInlining)]
48             protected override bool LessOrEqualThan(uint first, uint second) => first <= second;
49
50             [MethodImpl(MethodImplOptions.AggressiveInlining)]
51             protected override bool LessThanZero(uint value) => false; // value < 0 is always false
52             ↳ for uint
53
54             [MethodImpl(MethodImplOptions.AggressiveInlining)]
55             protected override bool LessThan(uint first, uint second) => first < second;
56
57             [MethodImpl(MethodImplOptions.AggressiveInlining)]
58             protected override uint Increment(uint value) => ++value;
59
60             [MethodImpl(MethodImplOptions.AggressiveInlining)]
61             protected override uint Decrement(uint value) => --value;
62
63             [MethodImpl(MethodImplOptions.AggressiveInlining)]
64             protected override uint Add(uint first, uint second) => first + second;
65
66             [MethodImpl(MethodImplOptions.AggressiveInlining)]
67             protected override uint Subtract(uint first, uint second) => first - second;
68
69             [MethodImpl(MethodImplOptions.AggressiveInlining)]
70             protected override bool FirstIsToTheLeftOfSecond(uint first, uint 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)]
79             protected override bool FirstIsToTheRightOfSecond(uint first, uint second)
80             {
81                 ref var firstLink = ref Links[first];
82             }
83         }
84     }
85 }

```

```

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 }

```

### 1.93 ./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
41             ↪ always true for uint
42
43             [MethodImpl(MethodImplOptions.AggressiveInlining)]
44             protected override bool LessOrEqualThanZero(uint value) => value == 0U; // value is
45             ↪ always >= 0 for uint
46
47             [MethodImpl(MethodImplOptions.AggressiveInlining)]
48             protected override bool LessOrEqualThan(uint first, uint second) => first <= second;
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,
71         ↪ secondLink.Source, secondLink.Target);
72 }
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected override bool FirstIsToTheRightOfSecond(uint first, uint second)
76 {
77     ref var firstLink = ref Links[first];
78     ref var secondLink = ref Links[second];
79     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
80         ↪ secondLink.Source, secondLink.Target);
81 }
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 protected override ref LinksHeader<uint> GetHeaderReference() => ref *Header;
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 protected override ref RawLink<uint> GetLinkReference(uint link) => ref Links[link];
88 }

```

## 1.94 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSourcesRecursionlessSizeBalancedTree

```

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 UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods :
8         ↪ UInt32LinksRecursionlessSizeBalancedTreeMethodsBase
9     {
10         public UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<uint>
11             ↪ constants, RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links,
12             ↪ header) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsSource;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected override ref uint GetRightReference(uint node) => ref
19             ↪ Links[node].RightAsSource;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override uint GetLeft(uint node) => Links[node].LeftAsSource;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override uint GetRight(uint node) => Links[node].RightAsSource;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override void SetLeft(uint node, uint left) => Links[node].LeftAsSource = left;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override void SetRight(uint node, uint right) => Links[node].RightAsSource =
32             ↪ right;
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected override uint GetSize(uint node) => Links[node].SizeAsSource;
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected override void SetSize(uint node, uint size) => Links[node].SizeAsSource = size;
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override uint GetTreeRoot() => Header->RootAsSource;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override uint GetBasePartValue(uint link) => Links[link].Source;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
48             ↪ uint secondSource, uint secondTarget)

```

```

43     => firstSource < secondSource || (firstSource == secondSource && firstTarget <
44         ↪ secondTarget);
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
48         ↪ uint secondSource, uint secondTarget)
49         => firstSource > secondSource || (firstSource == secondSource && firstTarget >
50             ↪ secondTarget);
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected override void ClearNode(uint node)
54     {
55         ref var link = ref Links[node];
56         link.LeftAsSource = 0U;
57         link.RightAsSource = 0U;
58         link.SizeAsSource = 0U;
59     }
60 }

```

## 1.95 ./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 :
8          ↪ UInt32LinksSizeBalancedTreeMethodsBase
9      {
10         public UInt32LinksSourcesSizeBalancedTreeMethods(LinksConstants<uint> constants,
11             ↪ RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links, header) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsSource;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected override ref uint GetRightReference(uint node) => ref
18             ↪ Links[node].RightAsSource;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override uint GetLeft(uint node) => Links[node].LeftAsSource;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override uint GetRight(uint node) => Links[node].RightAsSource;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override void SetLeft(uint node, uint left) => Links[node].LeftAsSource = left;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override void SetRight(uint node, uint right) => Links[node].RightAsSource =
31             ↪ right;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override uint GetSize(uint node) => Links[node].SizeAsSource;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override void SetSize(uint node, uint size) => Links[node].SizeAsSource = size;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override uint GetTreeRoot() => Header->RootAsSource;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override uint GetBasePartValue(uint link) => Links[link].Source;
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
47             ↪ uint secondSource, uint secondTarget)
48             => firstSource < secondSource || (firstSource == secondSource && firstTarget <
49                 ↪ secondTarget);
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
53             ↪ uint secondSource, uint secondTarget)
54             => firstSource > secondSource || (firstSource == secondSource && firstTarget >
55                 ↪ secondTarget);
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         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 }

```

## 1.96 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksTargetsRecursionlessSizeBalancedTree

```

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 UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods :
8          ↳ UInt32LinksRecursionlessSizeBalancedTreeMethodsBase
9      {
10
11         public UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<uint>
12             ↳ constants, RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links,
13             ↳ header) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref uint GetRightReference(uint node) => ref
20             ↳ Links[node].RightAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override uint GetLeft(uint node) => Links[node].LeftAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override uint GetRight(uint node) => Links[node].RightAsTarget;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(uint node, uint left) => Links[node].LeftAsTarget = left;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetRight(uint node, uint right) => Links[node].RightAsTarget =
33             ↳ right;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override uint GetSize(uint node) => Links[node].SizeAsTarget;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override void SetSize(uint node, uint size) => Links[node].SizeAsTarget = size;
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected override uint GetTreeRoot() => Header->RootAsTarget;
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override uint GetBasePartValue(uint link) => Links[link].Target;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
49             ↳ uint secondSource, uint secondTarget)
50             => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
51             ↳ secondSource);
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
55             ↳ uint secondSource, uint secondTarget)
56             => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
57             ↳ secondSource);
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override void ClearNode(uint node)
61         {
62             ref var link = ref Links[node];
63             link.LeftAsTarget = 0U;
64             link.RightAsTarget = 0U;
65             link.SizeAsTarget = 0U;
66         }
67     }
68 }

```

## 1.97 ./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 :
8          ↳ UInt32LinksSizeBalancedTreeMethodsBase
9      {
10         public UInt32LinksTargetsSizeBalancedTreeMethods(LinksConstants<uint> constants,
11             ↳ RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links, header) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsTarget;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected override ref uint GetRightReference(uint node) => ref
18             ↳ Links[node].RightAsTarget;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override uint GetLeft(uint node) => Links[node].LeftAsTarget;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override uint GetRight(uint node) => Links[node].RightAsTarget;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override void SetLeft(uint node, uint left) => Links[node].LeftAsTarget = left;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override void SetRight(uint node, uint right) => Links[node].RightAsTarget =
31             ↳ right;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override uint GetSize(uint node) => Links[node].SizeAsTarget;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override void SetSize(uint node, uint size) => Links[node].SizeAsTarget = size;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override uint GetTreeRoot() => Header->RootAsTarget;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override uint GetBasePartValue(uint link) => Links[link].Target;
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
47             ↳ uint secondSource, uint secondTarget)
48             => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
49                 ↳ secondSource);
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
53             ↳ uint secondSource, uint secondTarget)
54             => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
55                 ↳ secondSource);
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         protected override void ClearNode(uint node)
59         {
60             ref var link = ref Links[node];
61             link.LeftAsTarget = 0U;
62             link.RightAsTarget = 0U;
63             link.SizeAsTarget = 0U;
64         }
65     }
66 }

```

## 1.98 ./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
    ↳ organizing the storage of links with addresses represented as <see cref="uint" />.</para>
13  /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
    ↳ размером, для организации хранения связей с адресами представленными в виде <see
    ↳ cref="uint" />.</para>
14  /// </summary>
15  public unsafe class UInt32UnitedMemoryLinks : UnitedMemoryLinksBase<uint>
16  {
17      private readonly Func<ILinksTreeMethods<uint>> _createSourceTreeMethods;
18      private readonly Func<ILinksTreeMethods<uint>> _createTargetTreeMethods;
19      private LinksHeader<uint>* _header;
20      private RawLink<uint>* _links;
21
22      [MethodImpl(MethodImplOptions.AggressiveInlining)]
23      public UInt32UnitedMemoryLinks(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 UInt32UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
    ↳ FileMappedResizableDirectMemory(address, memoryReservationStep),
    ↳ memoryReservationStep) { }
32
33      [MethodImpl(MethodImplOptions.AggressiveInlining)]
34      public UInt32UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
    ↳ DefaultLinksSizeStep) { }
35
36      [MethodImpl(MethodImplOptions.AggressiveInlining)]
37      public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
    ↳ memoryReservationStep) : this(memory, memoryReservationStep,
    ↳ Default<LinksConstants<uint>>.Instance, IndexTreeType.Default) { }
38
39      [MethodImpl(MethodImplOptions.AggressiveInlining)]
40      public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
    ↳ memoryReservationStep, LinksConstants<uint> constants, IndexTreeType indexTreeType)
    ↳ : base(memory, memoryReservationStep, constants)
41      {
42          if (indexTreeType == IndexTreeType.SizeBalancedTree)
43          {
44              _createSourceTreeMethods = () => new
    ↳ UInt32LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
45              _createTargetTreeMethods = () => new
    ↳ UInt32LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
46          }
47          else
48          {
49              _createSourceTreeMethods = () => new
    ↳ UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods(Constants, _links,
    ↳ _header);
50              _createTargetTreeMethods = () => new
    ↳ UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods(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<uint>*)memory.Pointer;
59          _links = (RawLink<uint>*)memory.Pointer;
60          SourcesTreeMethods = _createSourceTreeMethods();
61          TargetsTreeMethods = _createTargetTreeMethods();
62          UnusedLinksListMethods = new UInt32UnusedLinksListMethods(_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<uint> GetHeaderReference() => ref *_header;
75
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     protected override ref RawLink<uint> GetLinkReference(uint linkIndex) => ref
78         ↪ _links[linkIndex];
79
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     protected override bool AreEqual(uint first, uint second) => first == second;
82
83     [MethodImpl(MethodImplOptions.AggressiveInlining)]
84     protected override bool LessThan(uint first, uint second) => first < second;
85
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     protected override bool LessOrEqualThan(uint first, uint second) => first <= second;
88
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     protected override bool GreaterThan(uint first, uint second) => first > second;
91
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     protected override bool GreaterOrEqualThan(uint first, uint second) => first >= second;
94
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     protected override uint GetZero() => 0U;
97
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     protected override uint GetOne() => 1U;
100
101     [MethodImpl(MethodImplOptions.AggressiveInlining)]
102     protected override long ConvertToInt64(uint value) => (long)value;
103
104     [MethodImpl(MethodImplOptions.AggressiveInlining)]
105     protected override uint ConvertToAddress(long value) => (uint)value;
106
107     [MethodImpl(MethodImplOptions.AggressiveInlining)]
108     protected override uint Add(uint first, uint second) => first + second;
109
110     [MethodImpl(MethodImplOptions.AggressiveInlining)]
111     protected override uint Subtract(uint first, uint second) => first - second;
112
113     [MethodImpl(MethodImplOptions.AggressiveInlining)]
114     protected override uint Increment(uint link) => ++link;
115
116     [MethodImpl(MethodImplOptions.AggressiveInlining)]
117     protected override uint Decrement(uint link) => --link;
118 }

```

## 1.99 ./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 }

```

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

```
1 using System.Runtime.CompilerServices;
2 using Platform.Data.Doublets.Memory.United.Generic;
3 using static System.Runtime.CompilerServices.Unsafe;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Memory.United.Specific
8 {
9     public unsafe abstract class UInt64LinksAvlBalancedTreeMethodsBase :
10         ↳ LinksAvlBalancedTreeMethodsBase<ulong>
11     {
12         protected new readonly RawLink<ulong>* Links;
13         protected new readonly LinksHeader<ulong>* Header;
14
15         protected UInt64LinksAvlBalancedTreeMethodsBase(LinksConstants<ulong> constants,
16             ↳ RawLink<ulong>* links, LinksHeader<ulong>* header)
17             : base(constants, (byte*)links, (byte*)header)
18         {
19             Links = links;
20             Header = header;
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override ulong GetZero() => OUL;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override bool EqualToZero(ulong value) => value == OUL;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override bool AreEqual(ulong first, ulong second) => first == second;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override bool GreaterThanZero(ulong value) => value > OUL;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override bool GreaterThan(ulong first, ulong second) => first > second;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
43             ↳ always true for ulong
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override bool LessOrEqualThanZero(ulong value) => value == OUL; // value is
47             ↳ always >= 0 for ulong
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
54             ↳ for ulong
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override bool LessThan(ulong first, ulong second) => first < second;
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override ulong Increment(ulong value) => ++value;
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         protected override ulong Decrement(ulong value) => --value;
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         protected override ulong Add(ulong first, ulong second) => first + second;
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         protected override ulong Subtract(ulong first, ulong second) => first - second;
70
71         [MethodImpl(MethodImplOptions.AggressiveInlining)]
72         protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
73         {
74             ref var firstLink = ref Links[first];
75             ref var secondLink = ref Links[second];
76             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
77                 ↳ secondLink.Source, secondLink.Target);
78         }
79     }
80 }
```

```

74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
76 {
77     ref var firstLink = ref Links[first];
78     ref var secondLink = ref Links[second];
79     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
80 }
81
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 protected override ulong GetSizeValue(ulong value) => (value & 4294967264UL) >> 5;
84
85 [MethodImpl(MethodImplOptions.AggressiveInlining)]
86 protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
    ↪ storedValue & 31UL | (size & 134217727UL) << 5;
87
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 protected override bool GetLeftIsChildValue(ulong value) => (value & 16UL) >> 4 == 1UL;
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
    ↪ storedValue = storedValue & 4294967279UL | (As<bool, byte>(ref value) & 1UL) << 4;
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 protected override bool GetRightIsChildValue(ulong value) => (value & 8UL) >> 3 == 1UL;
96
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>
    ↪ storedValue = storedValue & 4294967287UL | (As<bool, byte>(ref value) & 1UL) << 3;
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 protected override sbyte GetBalanceValue(ulong value) => unchecked((sbyte)(value & 7UL |
    ↪ 0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of
    ↪ sbyte
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 protected override void SetBalanceValue(ref ulong storedValue, sbyte value) =>
    ↪ storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
    ↪ value & 3) & 7UL);
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
108
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
111 }
112 }

```

## 1.101 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksRecursionlessSizeBalancedTreeMeth

```

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 UInt64LinksRecursionlessSizeBalancedTreeMethodsBase :
    ↪ LinksRecursionlessSizeBalancedTreeMethodsBase<ulong>
9     {
10         protected new readonly RawLink<ulong>* Links;
11         protected new readonly LinksHeader<ulong>* Header;
12
13         protected UInt64LinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<ulong>
    ↪ constants, RawLink<ulong>* links, LinksHeader<ulong>* header)
14             : base(constants, (byte*)links, (byte*)header)
15         {
16             Links = links;
17             Header = header;
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override ulong GetZero() => 0UL;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override bool EqualToZero(ulong value) => value == 0UL;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override bool AreEqual(ulong first, ulong second) => first == second;
28

```

```

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

```

## 1.102 ./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     protected UInt64LinksSizeBalancedTreeMethodsBase(LinksConstants<ulong> constants,
14         ↳ RawLink<ulong>* links, LinksHeader<ulong>* header)
15         : base(constants, (byte*)links, (byte*)header)
16     {
17         Links = links;
18         Header = header;
19     }
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     protected override ulong GetZero() => 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)]
79     protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
80     {
81         ref var firstLink = ref Links[first];
82         ref var secondLink = ref Links[second];
83         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
84             ↳ secondLink.Source, secondLink.Target);
85     }
86
87     [MethodImpl(MethodImplOptions.AggressiveInlining)]
88     protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
89
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];

```



```
86     }
87 }
```

1.103 ./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)]
62         protected override void SetRightIsChild(ulong node, bool value) =>
63         ↪ SetRightIsChildValue(ref Links[node].SizeAsSource, value);
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         protected override sbyte GetBalance(ulong node) =>
67         ↪ GetBalanceValue(Links[node].SizeAsSource);
68
69         [MethodImpl(MethodImplOptions.AggressiveInlining)]
70         protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
71         ↪ Links[node].SizeAsSource, value);
72
73         [MethodImpl(MethodImplOptions.AggressiveInlining)]
74         protected override ulong GetTreeRoot() => Header->RootAsSource;
75     }
76 }
```

```

62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
67     ↳ ulong secondSource, ulong secondTarget)
68     => firstSource < secondSource || (firstSource == secondSource && firstTarget <
69     ↳ secondTarget);
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
73     ↳ ulong secondSource, ulong secondTarget)
74     => firstSource > secondSource || (firstSource == secondSource && firstTarget >
75     ↳ secondTarget);
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected override void ClearNode(ulong node)
79 {
80     ref var link = ref Links[node];
81     link.LeftAsSource = OUL;
82     link.RightAsSource = OUL;
83     link.SizeAsSource = OUL;
84 }
85 }

```

#### 1.104 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods

```

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 UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods :
8     ↳ UInt64LinksRecursionlessSizeBalancedTreeMethodsBase
9     {
10         public UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<ulong>
11     ↳ constants, RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants,
12     ↳ links, header) { }
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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsSource =
41     ↳ size;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override ulong GetTreeRoot() => Header->RootAsSource;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
51     ↳ ulong secondSource, ulong secondTarget)

```

```

43     => firstSource < secondSource || (firstSource == secondSource && firstTarget <
44         ↪ secondTarget);
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
48         ↪ ulong secondSource, ulong secondTarget)
49         => firstSource > secondSource || (firstSource == secondSource && firstTarget >
50             ↪ secondTarget);
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected override void ClearNode(ulong node)
54     {
55         ref var link = ref Links[node];
56         link.LeftAsSource = OUL;
57         link.RightAsSource = OUL;
58         link.SizeAsSource = OUL;
59     }
60 }

```

#### 1.105 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.c

```

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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsSource =
41             ↪ size;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override ulong GetTreeRoot() => Header->RootAsSource;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
51             ↪ ulong secondSource, ulong secondTarget)
52             => firstSource < secondSource || (firstSource == secondSource && firstTarget <
53                 ↪ secondTarget);
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
57             ↪ ulong secondSource, ulong secondTarget)

```

```

47         => firstSource > secondSource || (firstSource == secondSource && firstTarget >
48             ↪ secondTarget);
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 protected override void ClearNode(ulong node)
52 {
53     ref var link = ref Links[node];
54     link.LeftAsSource = OUL;
55     link.RightAsSource = OUL;
56     link.SizeAsSource = OUL;
57 }
58 }

```

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

```

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

```

```

51     protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
    ↳ Links[node].SizeAsTarget, value);
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override ulong GetTreeRoot() => Header->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,
    ↳ ulong secondSource, ulong secondTarget)
61     => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
    ↳ secondSource);
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
    ↳ ulong secondSource, ulong secondTarget)
65     => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
    ↳ secondSource);
66
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     protected override void ClearNode(ulong node)
69     {
70         ref var link = ref Links[node];
71         link.LeftAsTarget = OUL;
72         link.RightAsTarget = OUL;
73         link.SizeAsTarget = OUL;
74     }
75 }
76 }

```

## 1.107 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods

```

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 UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↳ UInt64LinksRecursionlessSizeBalancedTreeMethodsBase
8      {
9          public UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods(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) => 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,
43         ↪     ulong secondSource, ulong secondTarget)
44         => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
45             ↪     secondSource);
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
49         ↪     ulong secondSource, ulong secondTarget)
50         => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
51             ↪     secondSource);
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override void ClearNode(ulong node)
55     {
56         ref var link = ref Links[node];
57         link.LeftAsTarget = OUL;
58         link.RightAsTarget = OUL;
59         link.SizeAsTarget = OUL;
60     }
61 }
62
63 }
64
65 }

```

1.108 ./.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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
30             ↳ left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
34             ↳ right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
41             ↳ size;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override ulong GetTreeRoot() => Header->RootAsTarget;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
51             ↳ ulong secondSource, ulong secondTarget)
52             ↳ => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
53                 ↳ secondSource);
54     }
55 }

```

```

44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
46     ↪     ulong secondSource, ulong secondTarget)
47     => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
48     ↪     secondSource);
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override void ClearNode(ulong node)
52     {
53         ref var link = ref Links[node];
54         link.LeftAsTarget = OUL;
55         link.RightAsTarget = OUL;
56         link.SizeAsTarget = OUL;
57     }
58 }

```

### 1.109 ./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
13     ↪     organizing the storage of links with addresses represented as <see cref="ulong"
14     ↪     />.</para>
15     /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
16     ↪     размером, для организации хранения связей с адресами представленными в виде <see
17     ↪     cref="ulong"/>.</para>
18     /// </summary>
19     public unsafe class UInt64UnitedMemoryLinks : UnitedMemoryLinksBase<ulong>
20     {
21         private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
22         private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
23         private LinksHeader<ulong>* _header;
24         private RawLink<ulong>* _links;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public UInt64UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
28
29         /// <summary>
30         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
31         ↪     минимальным шагом расширения базы данных.
32         /// </summary>
33         /// <param name="address">Полный путь к файлу базы данных.</param>
34         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
35         ↪     байтах.</param>
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public UInt64UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
38         ↪     FileMappedResizableDirectMemory(address, memoryReservationStep),
39         ↪     memoryReservationStep) { }
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public UInt64UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
43         ↪     DefaultLinksSizeStep) { }
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
47         ↪     memoryReservationStep) : this(memory, memoryReservationStep,
48         ↪     Default<LinksConstants<ulong>>.Instance, IndexTreeType.Default) { }
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
52         ↪     memoryReservationStep, LinksConstants<ulong> constants, IndexTreeType indexTreeType)
53         ↪     : base(memory, memoryReservationStep, constants)
54         {
55             if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
56             {
57                 _createSourceTreeMethods = () => new
58                 ↪     UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
59                 _createTargetTreeMethods = () => new
60                 ↪     UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);

```

```

46     }
47     else if (indexTreeType == IndexTreeType.SizeBalancedTree)
48     {
49         _createSourceTreeMethods = () => new
50             ↳ UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
51         _createTargetTreeMethods = () => new
52             ↳ UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
53     }
54     else
55     {
56         _createSourceTreeMethods = () => new
57             ↳ UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods(Constants, _links,
58             ↳ _header);
59         _createTargetTreeMethods = () => new
60             ↳ UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods(Constants, _links,
61             ↳ _header);
62     }
63     Init(memory, memoryReservationStep);
64 }
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 protected override void SetPointers(IResizableDirectMemory memory)
68 {
69     _header = (LinksHeader<ulong>*)memory.Pointer;
70     _links = (RawLink<ulong>*)memory.Pointer;
71     SourcesTreeMethods = _createSourceTreeMethods();
72     TargetsTreeMethods = _createTargetTreeMethods();
73     UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
74 }
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 protected override void ResetPointers()
78 {
79     base.ResetPointers();
80     _links = null;
81     _header = null;
82 }
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
89     ↳ _links[linkIndex];
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 protected override bool AreEqual(ulong first, ulong second) => first == second;
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 protected override bool LessThan(ulong first, ulong second) => first < second;
96
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 protected override bool GreaterThan(ulong first, ulong second) => first > second;
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 protected override ulong GetZero() => 0UL;
108
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 protected override ulong GetOne() => 1UL;
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 protected override long ConvertToInt64(ulong value) => (long)value;
114
115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 protected override ulong ConvertToAddress(long value) => (ulong)value;
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 protected override ulong Add(ulong first, ulong second) => first + second;
120
121 [MethodImpl(MethodImplOptions.AggressiveInlining)]
122 protected override ulong Subtract(ulong first, ulong second) => first - second;
123
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

118         protected override ulong Increment(ulong link) => ++link;
119
120         [MethodImpl(MethodImplOptions.AggressiveInlining)]
121         protected override ulong Decrement(ulong link) => --link;
122     }
123 }

```

#### 1.110 ./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 }

```

#### 1.111 ./csharp/Platform.Data.Doublets/Numbers/Raw/LongRawNumberSequenceToNumberConverter.cs

```

1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Stacks;
3  using Platform.Converters;
4  using Platform.Numbers;
5  using Platform.Reflection;
6  using Platform.Data.Doublets.Decorators;
7  using Platform.Data.Doublets.Sequences.Walkers;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11  namespace Platform.Data.Doublets.Numbers.Raw
12  {
13      public class LongRawNumberSequenceToNumberConverter<TSource, TTarget> :
14          ↳ LinksDecoratorBase<TSource>, IConverter<TSource, TTarget>
15      {
16         private static readonly int _bitsPerRawNumber = NumericType<TSource>.BitsSize - 1;
17         private static readonly unchecked_converter<TSource, TTarget> _sourceToTargetConverter =
18             ↳ unchecked_converter<TSource, TTarget>.Default;
19
20         private readonly IConverter<TSource> _numberToAddressConverter;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public LongRawNumberSequenceToNumberConverter(ILinks<TSource> links, IConverter<TSource>
24             ↳ numberToAddressConverter) : base(links) => _numberToAddressConverter =
25             ↳ numberToAddressConverter;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public TTarget Convert(TSource source)
29         {
30             var constants = Links.Constants;
31             var externalReferencesRange = constants.ExternalReferencesRange;
32             if (externalReferencesRange.HasValue &&
33                 ↳ externalReferencesRange.Value.Contains(source))
34             {
35                 return
36                     ↳ _sourceToTargetConverter.Convert(_numberToAddressConverter.Convert(source));
37             }
38             else
39             {
40                 var pair = Links.GetLink(source);
41                 var walker = new LeftSequenceWalker<TSource>(Links, new DefaultStack<TSource>(),
42                     ↳ (link) => externalReferencesRange.HasValue &&
43                     ↳ externalReferencesRange.Value.Contains(link));

```

```

36         TTarget result = default;
37         foreach (var element in walker.Walk(source))
38         {
39             result = Bit.Or(Bit.ShiftLeft(result, _bitsPerRawNumber), Convert(element));
40         }
41         return result;
42     }
43 }
44 }
45 }

```

#### 1.112 ./csharp/Platform.Data.Doublets/Numbers/Raw/NumberToLongRawNumberSequenceConverter.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Converters;
4  using Platform.Numbers;
5  using Platform.Reflection;
6  using Platform.Data.Doublets.Decorators;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Numbers.Raw
11 {
12     public class NumberToLongRawNumberSequenceConverter<TSource, TTarget> :
13         ↳ LinksDecoratorBase<TTarget>, IConverter<TSource, TTarget>
14     {
15         private static readonly Comparer<TSource> _comparer = Comparer<TSource>.Default;
16         private static readonly TSource _maximumValue = NumericType<TSource>.MaxValue;
17         private static readonly int _bitsPerRawNumber = NumericType<TTarget>.BitsSize - 1;
18         private static readonly TSource _bitMask = Bit.ShiftRight(_maximumValue,
19             ↳ NumericType<TTarget>.BitsSize + 1);
20         private static readonly TSource _maximumConvertibleAddress = CheckedConverter<TTarget,
21             ↳ TSource>.Default.Convert(Arithmetic.Decrement(Hybrid<TTarget>.ExternalZero));
22         private static readonly UncheckedConverter<TSource, TTarget> _sourceToTargetConverter =
23             ↳ UncheckedConverter<TSource, TTarget>.Default;
24
25         private readonly IConverter<TTarget> _addressToNumberConverter;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public NumberToLongRawNumberSequenceConverter(ILinks<TTarget> links, IConverter<TTarget>
29             ↳ addressToNumberConverter) : base(links) => _addressToNumberConverter =
30             ↳ addressToNumberConverter;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public TTarget Convert(TSource source)
34         {
35             if (_comparer.Compare(source, _maximumConvertibleAddress) > 0)
36             {
37                 var numberPart = Bit.And(source, _bitMask);
38                 var convertedNumber = _addressToNumberConverter.Convert(_sourceToTargetConverter
39                     ↳ .Convert(numberPart));
40                 return Links.GetOrCreate(convertedNumber, Convert(Bit.ShiftRight(source,
41                     ↳ _bitsPerRawNumber)));
42             }
43             else
44             {
45                 return
46                     ↳ _addressToNumberConverter.Convert(_sourceToTargetConverter.Convert(source));
47             }
48         }
49     }
50 }
51 }

```

#### 1.113 ./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;

```

```

15     private static readonly TLink _one = Arithmetic.Increment(_zero);
16
17     private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
        ↪ powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
        ↪ powerOf2ToUnaryNumberConverter;
21
22     [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 <
        ↪ NumericType<TLink>.BitsSize; i++)
29         {
30             if (_equalityComparer.Equals(Bit.And(number, _one), _one))
31             {
32                 target = _equalityComparer.Equals(target, nullConstant)
33                     ? _powerOf2ToUnaryNumberConverter.Convert(i)
34                     : links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
35             }
36             number = Bit.ShiftRight(number, 1);
37         }
38         return target;
39     }
40 }
41 }

```

#### 1.114 ./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>,
        ↪ IConverter<Doublet<TLink>, TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;
14
15         private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
16         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public LinkToItsFrequencyNumberConverter(
20             ILinks<TLink> links,
21             IProperty<TLink, TLink> frequencyPropertyOperator,
22             IConverter<TLink> unaryNumberToAddressConverter)
23             : base(links)
24         {
25             _frequencyPropertyOperator = frequencyPropertyOperator;
26             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public TLink Convert(Doublet<TLink> doublet)
31         {
32             var links = _links;
33             var link = links.SearchOrDefault(doublet.Source, doublet.Target);
34             if (_equalityComparer.Equals(link, default))
35             {
36                 throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
37             }
38             var frequency = _frequencyPropertyOperator.Get(link);
39             if (_equalityComparer.Equals(frequency, default))
40             {
41                 return default;
42             }
43             var frequencyNumber = links.GetSource(frequency);
44             return _unaryNumberToAddressConverter.Convert(frequencyNumber);
45         }
46     }

```

```
47 }
```

### 1.115 ./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 }
```

### 1.116 ./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             // Implementation of Convert method
37         }
38     }
39 }
```

```

31 {
32     if (_equalityComparer.Equals(unaryNumber, default))
33     {
34         return default;
35     }
36     if (_equalityComparer.Equals(unaryNumber, _unaryOne))
37     {
38         return _one;
39     }
40     var links = _links;
41     var source = links.GetSource(unaryNumber);
42     var target = links.GetTarget(unaryNumber);
43     if (_equalityComparer.Equals(source, target))
44     {
45         return _unaryToUInt64[unaryNumber];
46     }
47     else
48     {
49         var result = _unaryToUInt64[source];
50         TLink lastValue;
51         while (!_unaryToUInt64.TryGetValue(target, out lastValue))
52         {
53             source = links.GetSource(target);
54             result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
55             target = links.GetTarget(target);
56         }
57         result = Arithmetic<TLink>.Add(result, lastValue);
58         return result;
59     }
60 }
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 private static Dictionary<TLink, TLink> CreateUnaryToUInt64Dictionary(ILinks<TLink>
    ↪ links, TLink unaryOne)
64 {
65     var unaryToUInt64 = new Dictionary<TLink, TLink>
66     {
67         { unaryOne, _one }
68     };
69     var unary = unaryOne;
70     var number = _one;
71     for (var i = 1; i < 64; i++)
72     {
73         unary = links.GetOrCreate(unary, unary);
74         number = Double(number);
75         unaryToUInt64.Add(unary, number);
76     }
77     return unaryToUInt64;
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 private static TLink Double(TLink number) =>
    ↪ _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
82 }
83 }

```

### 1.117 ./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 }

```

### 1.118 ./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 }

```

### 1.119 ./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)]

```

```

59     private TLink GetValue(TLink container)=> _equalityComparer.Equals(container, default)
        ↳ ? default : _links.GetTarget(container);
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     public void Set(TLink link, TLink value)
63     {
64         var links = _links;
65         var property = links.GetOrCreate(link, _propertyMarker);
66         var container = GetContainer(property);
67         if (_equalityComparer.Equals(container, default))
68         {
69             links.GetOrCreate(property, value);
70         }
71         else
72         {
73             links.Update(container, property, value);
74         }
75     }
76 }
77 }

```

### 1.120 ./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 }

## 1.121 ./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         }
80     }
81 }
```

```

68     _minFrequencyToCompress = minFrequencyToCompress;
69     _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
70     ResetMaxDoublet();
71 }
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 public override TLink Convert(ICollection<TLink> source) =>
75     ↪ _baseConverter.Convert(Compress(source));
76
77 /// <remarks>
78 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
79 /// Faster version (doublets' frequencies dictionary is not recreated).
80 /// </remarks>
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 private ICollection<TLink> Compress(ICollection<TLink> sequence)
83 {
84     if (sequence.IsNullOrEmpty())
85     {
86         return null;
87     }
88     if (sequence.Count == 1)
89     {
90         return sequence;
91     }
92     if (sequence.Count == 2)
93     {
94         return new[] { _links.GetOrCreate(sequence[0], sequence[1]) };
95     }
96     // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
97     var copy = new HalfDoublet[sequence.Count];
98     Doublet<TLink> doublet = default;
99     for (var i = 1; i < sequence.Count; i++)
100     {
101         doublet = new Doublet<TLink>(sequence[i - 1], sequence[i]);
102         LinkFrequency<TLink> data;
103         if (_doInitialFrequenciesIncrement)
104         {
105             data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
106         }
107         else
108         {
109             data = _doubletFrequenciesCache.GetFrequency(ref doublet);
110             if (data == null)
111             {
112                 throw new NotSupportedException("If you ask not to increment
113                 ↪ frequencies, it is expected that all frequencies for the sequence
114                 ↪ are prepared.");
115             }
116         }
117         copy[i - 1].Element = sequence[i - 1];
118         copy[i - 1].DoubletData = data;
119         UpdateMaxDoublet(ref doublet, data);
120     }
121     copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
122     copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
123     if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
124     {
125         var newLength = ReplaceDoublets(copy);
126         sequence = new TLink[newLength];
127         for (int i = 0; i < newLength; i++)
128         {
129             sequence[i] = copy[i].Element;
130         }
131     }
132     return sequence;
133 }
134
135 /// <remarks>
136 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
137 /// </remarks>
138 [MethodImpl(MethodImplOptions.AggressiveInlining)]
139 private int ReplaceDoublets(HalfDoublet[] copy)
140 {
141     var oldLength = copy.Length;
142     var newLength = copy.Length;
143     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
144     {
145         var maxDoubletSource = _maxDoublet.Source;
146         var maxDoubletTarget = _maxDoublet.Target;

```

```

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

```

```

213         if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
214             (_comparer.Compare(maxFrequency, frequency) < 0 ||
                (_equalityComparer.Equals(maxFrequency, frequency) &&
                ↪ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
                ↪ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
                ↪ better stability and better compression on sequent data and even on random
                ↪ numbers data (but gives collisions anyway) */
215     {
216         _maxDoublet = doublet;
217         _maxDoubletData = data;
218     }
219 }
220 }
221 }

```

### 1.122 ./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     }

```

### 1.123 ./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             {

```

```

37         return sequence[0];
38     }
39     if (length == 2)
40     {
41         return _links.GetOrCreate(sequence[0], sequence[1]);
42     }
43     sequence = sequence.ToArray();
44     var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
45     while (length > 2)
46     {
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     }
106     return _links.GetOrCreate(sequence[0], sequence[1]);
107 }
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 private static TLink GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink previous, TLink
110     ↪ current, TLink next)
111 {
112     return _comparer.Compare(previous, next) > 0
113         ? _comparer.Compare(previous, current) < 0 ? previous : current
114         : _comparer.Compare(next, current) < 0 ? next : current;

```

```

113     }
114
115     [MethodImpl(MethodImplOptions.AggressiveInlining)]
116     private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
        ↪ _comparer.Compare(next, current) < 0 ? next : current;
117
118     [MethodImpl(MethodImplOptions.AggressiveInlining)]
119     private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
        ↪ => _comparer.Compare(previous, current) < 0 ? previous : current;
120 }
121 }

```

#### 1.124 ./csharp/Platform.Data.Doublets/Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs

```

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

```

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

```

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

```

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

```

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

```

```

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

```

### 1.127 ./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            {

```

```

49         right = links.GetOrCreate(left, right);
50         left = cursor;
51     }
52     return links.GetOrCreate(left, right);
53 }
54 }
55 }

```

### 1.128 ./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 }

```

### 1.129 ./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     }
49 }

```



```

38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
    ↪     (_listComparer.GetHashCode(pair.Key),
    ↪     _listComparer.GetHashCode(pair.Value)).GetHashCode();
40 }
41
42 private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
43 {
44     private readonly IListComparer<TLink> _listComparer;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
    ↪     KeyValuePair<IList<TLink>, IList<TLink>> right)
51     {
52         var intermediateResult = _listComparer.Compare(left.Key, right.Key);
53         if (intermediateResult == 0)
54         {
55             intermediateResult = _listComparer.Compare(left.Value, right.Value);
56         }
57         return intermediateResult;
58     }
59 }
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 public DuplicateSegmentsProvider(IList<TLink> links, IList<TLink> sequences)
63     : base(minimumStringSegmentLength: 2)
64 {
65     _links = links;
66     _sequences = sequences;
67 }
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
71 {
72     _groups = new HashSet<KeyValuePair<IList<TLink>,
    ↪     IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
73     var links = _links;
74     var count = links.Count();
75     _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
76     links.Each(link =>
77     {
78         var linkIndex = links.GetIndex(link);
79         var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
80         var constants = links.Constants;
81         if (!_visited.Get(linkBitIndex))
82         {
83             var sequenceElements = new List<TLink>();
84             var filler = new ListFiller<TLink, TLink>(sequenceElements, constants.Break);
85             _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
    ↪             LinkAddress<TLink>(linkIndex));
86             if (sequenceElements.Count > 2)
87             {
88                 WalkAll(sequenceElements);
89             }
90         }
91         return constants.Continue;
92     });
93     var resultList = _groups.ToList();
94     var comparer = Default<ItemComparer>.Instance;
95     resultList.Sort(comparer);
96     #if DEBUG
97     foreach (var item in resultList)
98     {
99         PrintDuplicates(item);
100     }
101     #endif
102     return resultList;
103 }
104
105 [MethodImpl(MethodImplOptions.AggressiveInlining)]
106 protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
    ↪     length) => new Segment<TLink>(elements, offset, length);
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 protected override void OnDuplicateFound(Segment<TLink> segment)
110 {

```

```

111     var duplicates = CollectDuplicatesForSegment(segment);
112     if (duplicates.Count > 1)
113     {
114         _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
115             ↪ duplicates));
116     }
117
118     [MethodImpl(MethodImplOptions.AggressiveInlining)]
119     private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
120     {
121         var duplicates = new List<TLink>();
122         var readAsElement = new HashSet<TLink>();
123         var restrictions = segment.ShiftRight();
124         var constants = _links.Constants;
125         restrictions[0] = constants.Any;
126         _sequences.Each(sequence =>
127         {
128             var sequenceIndex = sequence[constants.IndexPart];
129             duplicates.Add(sequenceIndex);
130             readAsElement.Add(sequenceIndex);
131             return constants.Continue;
132         }, restrictions);
133         if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
134         {
135             return new List<TLink>();
136         }
137         foreach (var duplicate in duplicates)
138         {
139             var duplicateBitIndex = _addressToInt64Converter.Convert(duplicate);
140             _visited.Set(duplicateBitIndex);
141         }
142         if (_sequences is Sequences sequencesExperiments)
143         {
144             var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
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 }

```

## 1.130 ./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(ICollection<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(ICollection<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         {

```

```

76     var number = GetFrequency(source, target).Frequency;
77     Console.WriteLine("{0},{1} - {2}", source, target, number);
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
82 {
83     if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
84     {
85         data.IncrementFrequency();
86     }
87     else
88     {
89         var link = _links.SearchOrDefault(doublet.Source, doublet.Target);
90         data = new LinkFrequency<TLink>(_one, link);
91         if (!_equalityComparer.Equals(link, default))
92         {
93             data.Frequency = Arithmetic.Add(data.Frequency,
94                 ↪ _frequencyCounter.Count(link));
95         }
96         _doubletsCache.Add(doublet, data);
97     }
98     return data;
99 }
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 public void ValidateFrequencies()
102 {
103     foreach (var entry in _doubletsCache)
104     {
105         var value = entry.Value;
106         var linkIndex = value.Link;
107         if (!_equalityComparer.Equals(linkIndex, default))
108         {
109             var frequency = value.Frequency;
110             var count = _frequencyCounter.Count(linkIndex);
111             // TODO: Why `frequency` always greater than `count` by 1?
112             if (((_comparer.Compare(frequency, count) > 0) &&
113                 ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count), _one) > 0))
114                 || ((_comparer.Compare(count, frequency) > 0) &&
115                 ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency), _one) > 0)))
116             {
117                 throw new InvalidOperationException("Frequencies validation failed.");
118             }
119             //else
120             //{
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 }

```

### 1.131 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs

```

1 using System.Runtime.CompilerServices;
2 using Platform.Numbers;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
7 {
8     public class LinkFrequency<TLink>
9     {
10         public TLink Frequency { get; set; }
11         public TLink Link { get; set; }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

14     public LinkFrequency(TLink frequency, TLink link)
15     {
16         Frequency = frequency;
17         Link = link;
18     }
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     public LinkFrequency() { }
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     public void IncrementFrequency() => Frequency = Arithmetic<TLink>.Increment(Frequency);
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public override string ToString() => $"F: {Frequency}, L: {Link}";
31 }
32 }

```

### 1.132 ./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 }

```

### 1.133 ./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 }

```

### 1.134 ./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         protected readonly ILinks<TLink> _links;
18         protected readonly TLink _sequenceLink;
19         protected readonly TLink _symbol;
20         protected TLink _total;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
24             ↳ TLink symbol)
25         {
26             _links = links;
27             _sequenceLink = sequenceLink;
28             _symbol = symbol;
29             _total = default;
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public virtual TLink Count()
34         {
35             if (_comparer.Compare(_total, default) > 0)
36             {
37                 return _total;
38             }
39             StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
40                 ↳ IsElement, VisitElement);
41             return _total;
42         }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
46             ↳ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
47             ↳ IsPartialPoint
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         private bool VisitElement(TLink element)
51         {
52             if (_equalityComparer.Equals(element, _symbol))
53             {
54                 _total = Arithmetic.Increment(_total);
55             }
56             return true;
57         }
58     }
59 }

```

### 1.135 ./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 }

```

```

22     }
23 }

```

### 1.136 ./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 }

```

### 1.137 ./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 }

```

### 1.138 ./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;

```

```

24     _symbol = symbol;
25     _visits = new HashSet<TLink>();
26     _total = default;
27 }
28
29 [MethodImpl(MethodImplOptions.AggressiveInlining)]
30 public TLink Count()
31 {
32     if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
33     {
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 }

```

### 1.139 ./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)

```



```

27     {
28         _heightPropertyMarker = heightPropertyMarker;
29         _baseHeightProvider = baseHeightProvider;
30         _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
31         _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
32         _propertyOperator = propertyOperator;
33     }
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     public TLink Get(TLink sequence)
37     {
38         TLink height;
39         var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
40         if (_equalityComparer.Equals(heightValue, default))
41         {
42             height = _baseHeightProvider.Get(sequence);
43             heightValue = _addressToUnaryNumberConverter.Convert(height);
44             _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
45         }
46         else
47         {
48             height = _unaryNumberToAddressConverter.Convert(heightValue);
49         }
50         return height;
51     }
52 }
53 }

```

#### 1.140 ./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 }

```

#### 1.141 ./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 }

```

#### 1.142 ./csharp/Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs

```

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

```

```

6
7 namespace Platform.Data.Doublets.Sequences.Indexes
8 {
9     public class CachedFrequencyIncrementingSequenceIndex<TLink> : ISequenceIndex<TLink>
10    {
11        private static readonly EqualityComparer<TLink> _equalityComparer =
12            ↳ EqualityComparer<TLink>.Default;
13
14        private readonly LinkFrequenciesCache<TLink> _cache;
15
16        [MethodImpl(MethodImplOptions.AggressiveInlining)]
17        public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
18            ↳ _cache = cache;
19
20        [MethodImpl(MethodImplOptions.AggressiveInlining)]
21        public bool Add(IList<TLink> sequence)
22        {
23            var indexed = true;
24            var i = sequence.Count;
25            while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
26                ↳ { }
27            for (; i >= 1; i--)
28            {
29                _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
30            }
31            return indexed;
32        }
33
34        [MethodImpl(MethodImplOptions.AggressiveInlining)]
35        private bool IsIndexedWithIncrement(TLink source, TLink target)
36        {
37            var frequency = _cache.GetFrequency(source, target);
38            if (frequency == null)
39            {
40                return false;
41            }
42            var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
43            if (indexed)
44            {
45                _cache.IncrementFrequency(source, target);
46            }
47            return indexed;
48        }
49
50        [MethodImpl(MethodImplOptions.AggressiveInlining)]
51        public bool MightContain(IList<TLink> sequence)
52        {
53            var indexed = true;
54            var i = sequence.Count;
55            while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
56            return indexed;
57        }
58
59        [MethodImpl(MethodImplOptions.AggressiveInlining)]
60        private bool IsIndexed(TLink source, TLink target)
61        {
62            var frequency = _cache.GetFrequency(source, target);
63            if (frequency == null)
64            {
65                return false;
66            }
67            return !_equalityComparer.Equals(frequency.Frequency, default);
68        }
69    }
70 }

```

### 1.143 ./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    {

```

```

12     private static readonly EqualityComparer<TLink> _equalityComparer =
13         ↳ EqualityComparer<TLink>.Default;
14
15     private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
16     private readonly IIncrementer<TLink> _frequencyIncrementer;
17
18     [MethodImpl(MethodImplOptions.AggressiveInlining)]
19     public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IProperty<TLink, TLink>
20         ↳ frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
21         : base(links)
22     {
23         _frequencyPropertyOperator = frequencyPropertyOperator;
24         _frequencyIncrementer = frequencyIncrementer;
25     }
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     public override bool Add(IList<TLink> sequence)
29     {
30         var indexed = true;
31         var i = sequence.Count;
32         while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
33             ↳ { }
34         for (; i >= 1; i--)
35         {
36             Increment(_links.GetOrCreate(sequence[i - 1], sequence[i]));
37         }
38         return indexed;
39     }
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     private bool IsIndexedWithIncrement(TLink source, TLink target)
43     {
44         var link = _links.SearchOrDefault(source, target);
45         var indexed = !_equalityComparer.Equals(link, default);
46         if (indexed)
47         {
48             Increment(link);
49         }
50         return indexed;
51     }
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     private void Increment(TLink link)
55     {
56         var previousFrequency = _frequencyPropertyOperator.Get(link);
57         var frequency = _frequencyIncrementer.Increment(previousFrequency);
58         _frequencyPropertyOperator.Set(link, frequency);
59     }
60 }

```

#### 1.144 ./csharp/Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Indexes
7 {
8     public interface ISequenceIndex<TLink>
9     {
10         /// <summary>
11         /// Индексирует последовательность глобально, и возвращает значение,
12         /// определяющие была ли запрошенная последовательность проиндексирована ранее.
13         /// </summary>
14         /// <param name="sequence">Последовательность для индексации.</param>
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         bool Add(IList<TLink> sequence);
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         bool MightContain(IList<TLink> sequence);
20     }
21 }

```

#### 1.145 ./csharp/Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs

```

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

```

```

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

```

#### 1.146 ./csharp/Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs

```

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

```

```

37         return indexed;
38     }
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     public bool MightContain(ICollection<TLink> sequence)
42     {
43         var links = _links.Unsync;
44         return _links.SyncRoot.ExecuteReadOperation(() =>
45         {
46             var indexed = true;
47             var i = sequence.Count;
48             while (--i >= 1 && (indexed =
49                 ↪ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
50                 ↪ sequence[i]), default))) { }
51             return indexed;
52         });
53     }
54 }

```

#### 1.147 ./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 }

```

#### 1.148 ./csharp/Platform.Data.Doublets/Sequences/Sequences.Experiments.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using System.Linq;
5  using System.Text;
6  using Platform.Collections;
7  using Platform.Collections.Sets;
8  using Platform.Collections.Stacks;
9  using Platform.Data.Exceptions;
10 using Platform.Data.Sequences;
11 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
12 using Platform.Data.Doublets.Sequences.Walkers;
13 using LinkIndex = System.UInt64;
14 using Stack = System.Collections.Generic.Stack<ulong>;
15
16 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
17
18 namespace Platform.Data.Doublets.Sequences
19 {
20     partial class Sequences
21     {
22         #region Create All Variants (Not Practical)
23
24         /// <remarks>
25         /// Number of links that is needed to generate all variants for
26         /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
27         /// </remarks>
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public ulong[] CreateAllVariants2(ulong[] sequence)
30         {
31             return _sync.ExecuteWriteOperation(() =>
32             {
33                 if (sequence.IsNullOrEmpty())
34                 {
35                     return Array.Empty<ulong>();
36                 }
37                 Links.EnsureLinkExists(sequence);
38                 if (sequence.Length == 1)
39                 {
40                     return sequence;
41                 }

```

```

42         return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
43     });
44 }
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
48 {
49     #if DEBUG
50         if ((stopAt - startAt) < 0)
51         {
52             throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
53                 ↳ меньше или равен stopAt");
54         }
55         #endif
56         if ((stopAt - startAt) == 0)
57         {
58             return new[] { sequence[startAt] };
59         }
60         if ((stopAt - startAt) == 1)
61         {
62             return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt]) };
63         }
64         var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
65         var last = 0;
66         for (var splitter = startAt; splitter < stopAt; splitter++)
67         {
68             var left = CreateAllVariants2Core(sequence, startAt, splitter);
69             var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
70             for (var i = 0; i < left.Length; i++)
71             {
72                 for (var j = 0; j < right.Length; j++)
73                 {
74                     var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
75                     if (variant == Constants.Null)
76                     {
77                         throw new NotImplementedException("Creation cancellation is not
78                             ↳ implemented.");
79                     }
80                     variants[last++] = variant;
81                 }
82             }
83         }
84         return variants;
85     }
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 public List<ulong> CreateAllVariants1(params ulong[] sequence)
89 {
90     return _sync.ExecuteWriteOperation(() =>
91     {
92         if (sequence.IsNullOrEmpty())
93         {
94             return new List<ulong>();
95         }
96         Links.Unsync.EnsureLinkExists(sequence);
97         if (sequence.Length == 1)
98         {
99             return new List<ulong> { sequence[0] };
100         }
101         var results = new
102             ↳ List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
103         return CreateAllVariants1Core(sequence, results);
104     });
105 }
106
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]
108 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
109 {
110     if (sequence.Length == 2)
111     {
112         var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
113         if (link == Constants.Null)
114         {
115             throw new NotImplementedException("Creation cancellation is not
116                 ↳ implemented.");
117         }
118         results.Add(link);
119         return results;
120     }

```

```

116     }
117     var innerSequenceLength = sequence.Length - 1;
118     var innerSequence = new ulong[innerSequenceLength];
119     for (var li = 0; li < innerSequenceLength; li++)
120     {
121         var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
122         if (link == Constants.Null)
123         {
124             throw new NotImplementedException("Creation cancellation is not
125                 ↪ implemented.");
126         }
127         for (var isi = 0; isi < li; isi++)
128         {
129             innerSequence[isi] = sequence[isi];
130         }
131         innerSequence[li] = link;
132         for (var isi = li + 1; isi < innerSequenceLength; isi++)
133         {
134             innerSequence[isi] = sequence[isi + 1];
135         }
136         CreateAllVariants1Core(innerSequence, results);
137     }
138     return results;
139 }
140 #endregion
141
142 [MethodImpl(MethodImplOptions.AggressiveInlining)]
143 public HashSet<ulong> Each1(params ulong[] sequence)
144 {
145     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
146     Each1(link =>
147     {
148         if (!visitedLinks.Contains(link))
149         {
150             visitedLinks.Add(link); // изучить почему случаются повторы
151         }
152         return true;
153     }, sequence);
154     return visitedLinks;
155 }
156
157 [MethodImpl(MethodImplOptions.AggressiveInlining)]
158 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
159 {
160     if (sequence.Length == 2)
161     {
162         Links.Unsync.Each(sequence[0], sequence[1], handler);
163     }
164     else
165     {
166         var innerSequenceLength = sequence.Length - 1;
167         for (var li = 0; li < innerSequenceLength; li++)
168         {
169             var left = sequence[li];
170             var right = sequence[li + 1];
171             if (left == 0 && right == 0)
172             {
173                 continue;
174             }
175             var linkIndex = li;
176             ulong[] innerSequence = null;
177             Links.Unsync.Each(doublet =>
178             {
179                 if (innerSequence == null)
180                 {
181                     innerSequence = new ulong[innerSequenceLength];
182                     for (var isi = 0; isi < linkIndex; isi++)
183                     {
184                         innerSequence[isi] = sequence[isi];
185                     }
186                     for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
187                     {
188                         innerSequence[isi] = sequence[isi + 1];
189                     }
190                 }
191                 innerSequence[linkIndex] = doublet[Constants.IndexPart];
192                 Each1(handler, innerSequence);
193                 return Constants.Continue;

```

```

194         }, Constants.Any, left, right);
195     }
196 }
197
198 [MethodImpl(MethodImplOptions.AggressiveInlining)]
199 public HashSet<ulong> EachPart(params ulong[] sequence)
200 {
201     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
202     EachPartCore(link =>
203     {
204         var linkIndex = link[Constants.IndexPart];
205         if (!visitedLinks.Contains(linkIndex))
206         {
207             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
208         }
209         return Constants.Continue;
210     }, sequence);
211     return visitedLinks;
212 }
213
214 [MethodImpl(MethodImplOptions.AggressiveInlining)]
215 public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
216 {
217     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
218     EachPartCore(link =>
219     {
220         var linkIndex = link[Constants.IndexPart];
221         if (!visitedLinks.Contains(linkIndex))
222         {
223             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
224             return handler(new LinkAddress<LinkIndex>(linkIndex));
225         }
226         return Constants.Continue;
227     }, sequence);
228 }
229
230 [MethodImpl(MethodImplOptions.AggressiveInlining)]
231 private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
232     ↪ sequence)
233 {
234     if (sequence.IsNullOrEmpty())
235     {
236         return;
237     }
238     Links.EnsureLinkIsAnyOrExists(sequence);
239     if (sequence.Length == 1)
240     {
241         var link = sequence[0];
242         if (link > 0)
243         {
244             handler(new LinkAddress<LinkIndex>(link));
245         }
246         else
247         {
248             Links.Each(Constants.Any, Constants.Any, handler);
249         }
250     }
251     else if (sequence.Length == 2)
252     {
253         //_links.Each(sequence[0], sequence[1], handler);
254         //  o_|      x_o ...
255         // x_|      |__|
256         Links.Each(sequence[1], Constants.Any, doublet =>
257         {
258             var match = Links.SearchOrDefault(sequence[0], doublet);
259             if (match != Constants.Null)
260             {
261                 handler(new LinkAddress<LinkIndex>(match));
262             }
263             return true;
264         });
265         // |_x      ... x_o
266         // |_o      |__|
267         Links.Each(Constants.Any, sequence[0], doublet =>
268         {
269             var match = Links.SearchOrDefault(doublet, sequence[1]);
270             if (match != 0)

```



```

271         {
272             handler(new LinkAddress<LinkIndex>(match));
273         }
274         return true;
275     });
276     //      .-x o-.
277     //      |___|
278     PartialStepRight(x => handler(x), sequence[0], sequence[1]);
279 }
280 else
281 {
282     throw new NotImplementedException();
283 }
284 }
285
286 [MethodImpl(MethodImplOptions.AggressiveInlining)]
287 private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
288 {
289     Links.Unsync.Each(Constants.Any, left, doublet =>
290     {
291         StepRight(handler, doublet, right);
292         if (left != doublet)
293         {
294             PartialStepRight(handler, doublet, right);
295         }
296         return true;
297     });
298 }
299
300 [MethodImpl(MethodImplOptions.AggressiveInlining)]
301 private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
302 {
303     Links.Unsync.Each(left, Constants.Any, rightStep =>
304     {
305         TryStepRightUp(handler, right, rightStep);
306         return true;
307     });
308 }
309
310 [MethodImpl(MethodImplOptions.AggressiveInlining)]
311 private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
312     ↪ stepFrom)
313 {
314     var upStep = stepFrom;
315     var firstSource = Links.Unsync.GetTarget(upStep);
316     while (firstSource != right && firstSource != upStep)
317     {
318         upStep = firstSource;
319         firstSource = Links.Unsync.GetSource(upStep);
320     }
321     if (firstSource == right)
322     {
323         handler(new LinkAddress<LinkIndex>(stepFrom));
324     }
325 }
326
327 // TODO: Test
328 [MethodImpl(MethodImplOptions.AggressiveInlining)]
329 private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
330 {
331     Links.Unsync.Each(right, Constants.Any, doublet =>
332     {
333         StepLeft(handler, left, doublet);
334         if (right != doublet)
335         {
336             PartialStepLeft(handler, left, doublet);
337         }
338         return true;
339     });
340 }
341
342 [MethodImpl(MethodImplOptions.AggressiveInlining)]
343 private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
344 {
345     Links.Unsync.Each(Constants.Any, right, leftStep =>
346     {
347         TryStepLeftUp(handler, left, leftStep);
348         return true;
349     });

```

```

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(IList<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)

```

```

        {
            filterPosition = -2; // Длиннее чем нужно
            return false;
        }
        if (x != sequence[filterPosition])
        {
            filterPosition = -1;
            return false; // Начинается иначе
        }
        filterPosition++;

        return true;
    });
    if (filterPosition == sequence.Length)
    {
        results.Add(resultIndex);
    }
}
if (sequence.Length >= 2)
{
    StepRight(handler, sequence[0], sequence[1]);
}
var last = sequence.Length - 2;
for (var i = 1; i < last; i++)
{
    PartialStepRight(handler, sequence[i], sequence[i + 1]);
}
if (sequence.Length >= 3)
{
    StepLeft(handler, sequence[sequence.Length - 2],
        ↪ sequence[sequence.Length - 1]);
}
}
return results;
});
});
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
{
    return _sync.ExecuteReadOperation(() =>
    {
        var results = new HashSet<ulong>();
        if (sequence.Length > 0)
        {
            Links.EnsureLinkExists(sequence);
            var firstElement = sequence[0];
            if (sequence.Length == 1)
            {
                results.Add(firstElement);
                return results;
            }
            if (sequence.Length == 2)
            {
                var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
                if (doublet != Constants.Null)
                {
                    results.Add(doublet);
                }
                return results;
            }
            var matcher = new Matcher(this, sequence, results, null);
            if (sequence.Length >= 2)
            {
                StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
            }
            var last = sequence.Length - 2;
            for (var i = 1; i < last; i++)
            {
                PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
                    ↪ sequence[i + 1]);
            }
            if (sequence.Length >= 3)
            {
                StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
                    ↪ sequence[sequence.Length - 1]);
            }
        }
    });
}

```

```

501         return results;
502     });
503 }
504
505 public const int MaxSequenceFormatSize = 200;
506
507 [MethodImpl(MethodImplOptions.AggressiveInlining)]
508 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
509     => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
510
511 [MethodImpl(MethodImplOptions.AggressiveInlining)]
512 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
513     => elementToString, bool insertComma, params LinkIndex[] knownElements) =>
514     Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
515         elementToString, insertComma, knownElements));
516
517 [MethodImpl(MethodImplOptions.AggressiveInlining)]
518 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
519     => Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
520     => LinkIndex[] knownElements)
521 {
522     var linksInSequence = new HashSet<ulong>(knownElements);
523     //var entered = new HashSet<ulong>();
524     var sb = new StringBuilder();
525     sb.Append('{');
526     if (links.Exists(sequenceLink))
527     {
528         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
529             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
530             => entered.AddAndReturnVoid, x => { }, entered.DoNotContains
531         {
532             if (insertComma && sb.Length > 1)
533             {
534                 sb.Append(',');
535             }
536             //if (entered.Contains(element))
537             //{
538                 sb.Append('{');
539                 elementToString(sb, element);
540                 sb.Append('}');
541             //}
542             //else
543             elementToString(sb, element);
544             if (sb.Length < MaxSequenceFormatSize)
545             {
546                 return true;
547             }
548             sb.Append(insertComma ? ", ..." : "...");
549             return false;
550         }
551     }
552     sb.Append('}');
553     return sb.ToString();
554 }
555
556 [MethodImpl(MethodImplOptions.AggressiveInlining)]
557 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
558     => knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
559     => knownElements);
560
561 [MethodImpl(MethodImplOptions.AggressiveInlining)]
562 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
563     => LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
564     Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
565         sequenceLink, elementToString, insertComma, knownElements));
566
567 [MethodImpl(MethodImplOptions.AggressiveInlining)]
568 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
569     => Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
570     => LinkIndex[] knownElements)
571 {
572     var linksInSequence = new HashSet<ulong>(knownElements);
573     var entered = new HashSet<ulong>();
574     var sb = new StringBuilder();
575     sb.Append('{');
576     if (links.Exists(sequenceLink))
577     {
578         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,

```

```

565 x => linksInSequence.Contains(x) || links.IsFullPoint(x),
566 ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
567 {
568     if (insertComma && sb.Length > 1)
569     {
570         sb.Append(',');
571     }
572     if (entered.Contains(element))
573     {
574         sb.Append('{');
575         elementToString(sb, element);
576         sb.Append('}');
577     }
578     else
579     {
580         elementToString(sb, element);
581     }
582     if (sb.Length < MaxSequenceFormatSize)
583     {
584         return true;
585     }
586     sb.Append(insertComma ? ", ..." : "...");
587     return false;
588 });
589 }
590 sb.Append('}');
591 return sb.ToString();
592 }
593 [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                 if (filterPosition == (sequence.Length - 1))
641                 {

```

```

640         filteredResults.Add(result);
641     }
642 }
643     return filteredResults;
644 }
645     return new List<ulong>();
646 });
647 }
648
649 [MethodImpl(MethodImplOptions.AggressiveInlining)]
650 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
651 {
652     return _sync.ExecuteReadOperation(() =>
653     {
654         if (sequence.Length > 0)
655         {
656             Links.EnsureLinkExists(sequence);
657             var results = new HashSet<ulong>();
658             for (var i = 0; i < sequence.Length; i++)
659             {
660                 AllUsagesCore(sequence[i], results);
661             }
662             var filteredResults = new HashSet<ulong>();
663             var matcher = new Matcher(this, sequence, filteredResults, null);
664             matcher.AddAllPartialMatchedToResults(results);
665             return filteredResults;
666         }
667         return new HashSet<ulong>();
668     });
669 }
670
671 [MethodImpl(MethodImplOptions.AggressiveInlining)]
672 public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
673     ↪ params ulong[] sequence)
674 {
675     return _sync.ExecuteReadOperation(() =>
676     {
677         if (sequence.Length > 0)
678         {
679             Links.EnsureLinkExists(sequence);
680
681             var results = new HashSet<ulong>();
682             var filteredResults = new HashSet<ulong>();
683             var matcher = new Matcher(this, sequence, filteredResults, handler);
684             for (var i = 0; i < sequence.Length; i++)
685             {
686                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
687                 {
688                     return false;
689                 }
690             }
691             return true;
692         }
693         return true;
694     });
695 }
696
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 //         var filteredResults = new HashSet<ulong>();
719 //         var matcher = new Matcher(this, sequence, filteredResults, null);
720 //         matcher.AddAllPartialMatchedToResults(firstResults);
721 //         return filteredResults;
722 //     }
723
724 //     return new HashSet<ulong>();
725 // });
726 //}
727
728 [MethodImpl(MethodImplOptions.AggressiveInlining)]
729 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
730 {
731     return _sync.ExecuteReadOperation((Func<HashSet<ulong>>)(() =>
732     {
733         if (sequence.Length > 0)
734         {
735             ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links,
736                 ↪ (IList<ulong>)sequence);
737             var firstResults = new HashSet<ulong>();
738             var lastResults = new HashSet<ulong>();
739             var first = sequence.First(x => x != Constants.Any);
740             var last = sequence.Last(x => x != Constants.Any);
741             AllUsagesCore(first, firstResults);
742             AllUsagesCore(last, lastResults);
743             firstResults.IntersectWith(lastResults);
744             //for (var i = 0; i < sequence.Length; i++)
745             //    AllUsagesCore(sequence[i], results);
746             var filteredResults = new HashSet<ulong>();
747             var matcher = new Matcher(this, sequence, filteredResults, null);
748             matcher.AddAllPartialMatchedToResults(firstResults);
749             return filteredResults;
750         }
751         return new HashSet<ulong>();
752     }));
753 }
754
755 [MethodImpl(MethodImplOptions.AggressiveInlining)]
756 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
757     ↪ IList<ulong> sequence)
758 {
759     return _sync.ExecuteReadOperation(() =>
760     {
761         if (sequence.Count > 0)
762         {
763             Links.EnsureLinkExists(sequence);
764             var results = new HashSet<LinkIndex>();
765             //var nextResults = new HashSet<ulong>();
766             //for (var i = 0; i < sequence.Length; i++)
767             //{
768             //    AllUsagesCore(sequence[i], nextResults);
769             //    if (results.IsNullOrEmpty())
770             //    {
771             //        results = nextResults;
772             //        nextResults = new HashSet<ulong>();
773             //    }
774             //    else
775             //    {
776             //        results.IntersectWith(nextResults);
777             //        nextResults.Clear();
778             //    }
779             //}
780             var collector1 = new AllUsagesCollector1(Links.Unsync, results);
781             collector1.Collect(Links.Unsync.GetLink(sequence[0]));
782             var next = new HashSet<ulong>();
783             for (var i = 1; i < sequence.Count; i++)
784             {
785                 var collector = new AllUsagesCollector1(Links.Unsync, next);
786                 collector.Collect(Links.Unsync.GetLink(sequence[i]));
787
788                 results.IntersectWith(next);
789                 next.Clear();
790             }
791             var filteredResults = new HashSet<ulong>();
792             var matcher = new Matcher(this, sequence, filteredResults, null,
793                 ↪ readAsElements);
794             matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
795                 ↪ x)); // OrderBy is a Hack

```

```

792         return filteredResults;
793     }
794     return new HashSet<ulong>();
795 });
796 }
797
798 // Does not work
799 //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
800 //    params ulong[] sequence)
801 //{
802 //    var visited = new HashSet<ulong>();
803 //    var results = new HashSet<ulong>();
804 //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
805 //        true; }, readAsElements);
806 //    var last = sequence.Length - 1;
807 //    for (var i = 0; i < last; i++)
808 //    {
809 //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
810 //    }
811 //    return results;
812 //}
813
814 [MethodImpl(MethodImplOptions.AggressiveInlining)]
815 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
816 {
817     return _sync.ExecuteReadOperation(() =>
818     {
819         if (sequence.Length > 0)
820         {
821             Links.EnsureLinkExists(sequence);
822             //var firstElement = sequence[0];
823             //if (sequence.Length == 1)
824             //{
825             //    //results.Add(firstElement);
826             //    return results;
827             //}
828             //if (sequence.Length == 2)
829             //{
830             //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
831             //    //if (doublet != Doublets.Links.Null)
832             //    //    results.Add(doublet);
833             //    return results;
834             //}
835             //var lastElement = sequence[sequence.Length - 1];
836             //Func<ulong, bool> handler = x =>
837             //{
838             //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
839             //        results.Add(x);
840             //    return true;
841             //};
842             //if (sequence.Length >= 2)
843             //    StepRight(handler, sequence[0], sequence[1]);
844             //var last = sequence.Length - 2;
845             //for (var i = 1; i < last; i++)
846             //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
847             //if (sequence.Length >= 3)
848             //    StepLeft(handler, sequence[sequence.Length - 2],
849             //        sequence[sequence.Length - 1]);
850             //if (sequence.Length == 1)
851             //    throw new NotImplementedException(); // all sequences, containing
852             //    this element?
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)

```



```

864         matches.Add(results);
865     }
866     else
867     {
868         return results;
869         if (matches.Count == 2)
870         {
871             var merged = new List<ulong>();
872             for (var j = 0; j < matches[0].Count; j++)
873                 for (var k = 0; k < matches[1].Count; k++)
874                     CloseInnerConnections(merged.Add, matches[0][j],
875                                             matches[1][k]);
876             if (merged.Count > 0)
877                 matches = new List<List<ulong>> { merged };
878             else
879                 return new List<ulong>();
880         }
881     }
882     if (matches.Count > 0)
883     {
884         var usages = new HashSet<ulong>();
885         for (int i = 0; i < sequence.Length; i++)
886         {
887             AllUsagesCore(sequence[i], usages);
888         }
889         //for (int i = 0; i < matches[0].Count; i++)
890         //    AllUsagesCore(matches[0][i], usages);
891         //usages.UnionWith(matches[0]);
892         return usages.ToList();
893     }
894     var firstLinkUsages = new HashSet<ulong>();
895     AllUsagesCore(sequence[0], firstLinkUsages);
896     firstLinkUsages.Add(sequence[0]);
897     //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
898     //    sequence[0] }; // or all sequences, containing this element?
899     //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
900     //    1).ToList();
901     var results = new HashSet<ulong>();
902     foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
903         firstLinkUsages, 1))
904     {
905         AllUsagesCore(match, results);
906     }
907     return results.ToList();
908 }
909 return new List<ulong>();
910 });
911 }
912
913 /// <remarks>
914 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
915 /// </remarks>
916 [MethodImpl(MethodImplOptions.AggressiveInlining)]
917 public HashSet<ulong> AllUsages(ulong link)
918 {
919     return _sync.ExecuteReadOperation(() =>
920     {
921         var usages = new HashSet<ulong>();
922         AllUsagesCore(link, usages);
923         return usages;
924     });
925 }
926
927 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
928 // той связи с которой начинался поиск (STTTSSSTT),
929 // причём достаточно одного бита для хранения перехода влево или вправо
930 [MethodImpl(MethodImplOptions.AggressiveInlining)]
931 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
932 {
933     bool handler(ulong doublet)
934     {
935         if (usages.Add(doublet))
936         {
937             AllUsagesCore(doublet, usages);
938         }
939         return true;
940     }
941     Links.Unsync.Each(link, Constants.Any, handler);
942     Links.Unsync.Each(Constants.Any, link, handler);
943 }

```

```

936 }
937
938 [MethodImpl(MethodImplOptions.AggressiveInlining)]
939 public HashSet<ulong> AllBottomUsages(ulong link)
940 {
941     return _sync.ExecuteReadOperation(() =>
942     {
943         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,
978             ↳ Options.MarkedSequenceMatcher, symbol);
979         return counter.Count();
980     }
981     else
982     {
983         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
984             ↳ symbol);
985         return counter.Count();
986     }
987 }
988
989 [MethodImpl(MethodImplOptions.AggressiveInlining)]
990 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
    ↳ LinkIndex> outerHandler)
991 {
992     bool handler(ulong doublet)
993     {
994         if (usages.Add(doublet))
995         {
996             if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
997             {
998                 return false;
999             }
1000             if (!AllUsagesCore1(doublet, usages, outerHandler))
1001             {
1002                 return false;
1003             }
1004         }
1005         return true;
1006     }
1007     return Links.Unsync.Each(link, Constants.Any, handler)
1008         && Links.Unsync.Each(Constants.Any, link, handler);
1009 }
1010
1011 [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,
1037         ↪ CalculateCore);
1038
1039     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1040     private bool CalculateCore(ulong link)
1041     {
1042         if (_totals[link] == 0)
1043         {
1044             var total = 1UL;
1045             _totals[link] = total;
1046             var visitedChildren = new HashSet<ulong>();
1047             bool linkCalculator(ulong child)
1048             {
1049                 if (link != child && visitedChildren.Add(child))
1050                 {
1051                     total += _totals[child] == 0 ? 1 : _totals[child];
1052                 }
1053                 return true;
1054             }
1055             _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1056             _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1057             _totals[link] = total;
1058         }
1059         return true;
1060     }
1061 }
1062
1063 private class AllUsagesCalculator2
1064 {
1065     private readonly SynchronizedLinks<ulong> _links;
1066     private readonly ulong[] _totals;
1067
1068     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1069     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1070     {
1071         _links = links;
1072         _totals = totals;
1073     }
1074
1075     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1076     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1077         ↪ CalculateCore);
1078
1079     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1080     private bool IsElement(ulong link)
1081     {
1082         // _links.InSequence.Contains(link) ||
1083         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1084             ↪ link;
1085     }
1086
1087     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1088     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 private class AllUsagesCollector
1149 {
1150     private readonly ILinks<ulong> _links;
1151     private readonly HashSet<ulong> _usages;
1152
1153     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1154     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1155     {
1156         _links = links;
1157         _usages = usages;
1158     }
1159
1160     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1161     public bool Collect(ulong link)
1162     {
1163         if (_usages.Add(link))
1164

```

```

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

```

```

1385 }
1386
1387 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1388 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1389 {
1390     return _sync.ExecuteReadOperation(() =>
1391     {
1392         var results = new HashSet<ulong>();
1393         if (linksToConnect.Length > 0)
1394         {
1395             Links.EnsureLinkExists(linksToConnect);
1396             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1397             collector1.Collect(linksToConnect[0]);
1398             var next = new HashSet<ulong>();
1399             for (var i = 1; i < linksToConnect.Length; i++)
1400             {
1401                 var collector = new AllUsagesCollector(Links.Unsync, next);
1402                 collector.Collect(linksToConnect[i]);
1403                 results.IntersectWith(next);
1404                 next.Clear();
1405             }
1406             return results;
1407         }
1408     });
1409 }
1410
1411 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1412 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1413 {
1414     return _sync.ExecuteReadOperation(() =>
1415     {
1416         var results = new HashSet<ulong>();
1417         if (linksToConnect.Length > 0)
1418         {
1419             Links.EnsureLinkExists(linksToConnect);
1420             var collector1 = new AllUsagesCollector(Links, results);
1421             collector1.Collect(linksToConnect[0]);
1422             //AllUsagesCore(linksToConnect[0], results);
1423             for (var i = 1; i < linksToConnect.Length; i++)
1424             {
1425                 var next = new HashSet<ulong>();
1426                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1427                 collector.Collect(linksToConnect[i]);
1428                 //AllUsagesCore(linksToConnect[i], next);
1429                 //results.IntersectWith(next);
1430                 results = next;
1431             }
1432             return results;
1433         }
1434     });
1435 }
1436
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)]

```



```

1461 private static ulong[] Simplify(ulong[] sequence)
1462 {
1463     // Считаем новый размер последовательности
1464     long newLength = 0;
1465     var zeroOrManyStepped = false;
1466     for (var i = 0; i < sequence.Length; i++)
1467     {
1468         if (sequence[i] == ZeroOrMany)
1469         {
1470             if (zeroOrManyStepped)
1471             {
1472                 continue;
1473             }
1474             zeroOrManyStepped = true;
1475         }
1476         else
1477         {
1478             //if (zeroOrManyStepped) Is it efficient?
1479             zeroOrManyStepped = false;
1480         }
1481         newLength++;
1482     }
1483     // Строим новую последовательность
1484     zeroOrManyStepped = false;
1485     var newSequence = new ulong[newLength];
1486     long j = 0;
1487     for (var i = 0; i < sequence.Length; i++)
1488     {
1489         //var current = zeroOrManyStepped;
1490         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1491         //if (current && zeroOrManyStepped)
1492         //    continue;
1493         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1494         //if (zeroOrManyStepped && newZeroOrManyStepped)
1495         //    continue;
1496         //zeroOrManyStepped = newZeroOrManyStepped;
1497         if (sequence[i] == ZeroOrMany)
1498         {
1499             if (zeroOrManyStepped)
1500             {
1501                 continue;
1502             }
1503             zeroOrManyStepped = true;
1504         }
1505         else
1506         {
1507             //if (zeroOrManyStepped) Is it efficient?
1508             zeroOrManyStepped = false;
1509         }
1510         newSequence[j++] = sequence[i];
1511     }
1512     return newSequence;
1513 }
1514
1515 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1516 public static void TestSimplify()
1517 {
1518     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
1519     ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1520     var simplifiedSequence = Simplify(sequence);
1521 }
1522
1523 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1524 public List<ulong> GetSimilarSequences() => new List<ulong>();
1525
1526 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1527 public void Prediction()
1528 {
1529     //_links
1530     //sequences
1531 }
1532
1533 #region From Triplets
1534
1535 //public static void DeleteSequence(Link sequence)
1536 //{
1537 //}
1538
1539 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

1539 public List<ulong> CollectMatchingSequences(ulong[] links)
1540 {
1541     if (links.Length == 1)
1542     {
1543         throw new InvalidOperationException("Подпоследовательности с одним элементом не
1544             ↳ поддерживаются.");
1545     }
1546     var leftBound = 0;
1547     var rightBound = links.Length - 1;
1548     var left = links[leftBound++];
1549     var right = links[rightBound--];
1550     var results = new List<ulong>();
1551     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1552     return results;
1553 }
1554 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1555 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1556     ↳ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1557 {
1558     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1559     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1560     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1561     {
1562         var nextLeftLink = middleLinks[leftBound];
1563         var elements = GetRightElements(leftLink, nextLeftLink);
1564         if (leftBound <= rightBound)
1565         {
1566             for (var i = elements.Length - 1; i >= 0; i--)
1567             {
1568                 var element = elements[i];
1569                 if (element != 0)
1570                 {
1571                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
1572                         ↳ rightLink, rightBound, ref results);
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 }

```

```

1613     }
1614
1615     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1616     public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1617     {
1618         var result = new ulong[5];
1619         TryStepRight(startLink, rightLink, result, 0);
1620         Links.Each(Constants.Any, startLink, couple =>
1621         {
1622             if (couple != startLink)
1623             {
1624                 if (TryStepRight(couple, rightLink, result, 2))
1625                 {
1626                     return false;
1627                 }
1628             }
1629             return true;
1630         });
1631         if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1632         {
1633             result[4] = startLink;
1634         }
1635         return result;
1636     }
1637
1638     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1639     public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1640     {
1641         var added = 0;
1642         Links.Each(startLink, Constants.Any, couple =>
1643         {
1644             if (couple != startLink)
1645             {
1646                 var coupleTarget = Links.GetTarget(couple);
1647                 if (coupleTarget == rightLink)
1648                 {
1649                     result[offset] = couple;
1650                     if (++added == 2)
1651                     {
1652                         return false;
1653                     }
1654                 }
1655                 else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
1656                     ↳ == Net.And &&
1657                 {
1658                     result[offset + 1] = couple;
1659                     if (++added == 2)
1660                     {
1661                         return false;
1662                     }
1663                 }
1664             }
1665             return true;
1666         });
1667         return added > 0;
1668     }
1669
1670     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1671     public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1672     {
1673         var result = new ulong[5];
1674         TryStepLeft(startLink, leftLink, result, 0);
1675         Links.Each(startLink, Constants.Any, couple =>
1676         {
1677             if (couple != startLink)
1678             {
1679                 if (TryStepLeft(couple, leftLink, result, 2))
1680                 {
1681                     return false;
1682                 }
1683             }
1684             return true;
1685         });
1686         if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1687         {
1688             result[4] = leftLink;
1689         }
1690         return result;
1691     }

```

```

1691 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1692 public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1693 {
1694     var added = 0;
1695     Links.Each(Constants.Any, startLink, couple =>
1696     {
1697         if (couple != startLink)
1698         {
1699             var coupleSource = Links.GetSource(couple);
1700             if (coupleSource == leftLink)
1701             {
1702                 result[offset] = couple;
1703                 if (++added == 2)
1704                 {
1705                     return false;
1706                 }
1707             }
1708             else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1709                 ↪ == Net.And &&
1710             {
1711                 result[offset + 1] = couple;
1712                 if (++added == 2)
1713                 {
1714                     return false;
1715                 }
1716             }
1717         }
1718         return true;
1719     });
1720     return added > 0;
1721 }
1722
1723 #endregion
1724
1725 #region Walkers
1726
1727 public class PatternMatcher : RightSequenceWalker<ulong>
1728 {
1729     private readonly Sequences _sequences;
1730     private readonly ulong[] _patternSequence;
1731     private readonly HashSet<LinkIndex> _linksInSequence;
1732     private readonly HashSet<LinkIndex> _results;
1733
1734     #region Pattern Match
1735
1736     enum PatternBlockType
1737     {
1738         Undefined,
1739         Gap,
1740         Elements
1741     }
1742
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)]

```

```

protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
    ↪ base.IsElement(link);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public bool PatternMatch(LinkIndex sequenceToMatch)
{
    _patternPosition = 0;
    _sequencePosition = 0;
    foreach (var part in Walk(sequenceToMatch))
    {
        if (!PatternMatchCore(part))
        {
            break;
        }
    }
    return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
    ↪ - 1 && _pattern[_patternPosition].Start == 0);
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
private List<PatternBlock> CreateDetailedPattern()
{
    var pattern = new List<PatternBlock>();
    var patternBlock = new PatternBlock();
    for (var i = 0; i < _patternSequence.Length; i++)
    {
        if (patternBlock.Type == PatternBlockType.Undefined)
        {
            if (_patternSequence[i] == _sequences.Constants.Any)
            {
                patternBlock.Type = PatternBlockType.Gap;
                patternBlock.Start = 1;
                patternBlock.Stop = 1;
            }
            else if (_patternSequence[i] == ZeroOrMany)
            {
                patternBlock.Type = PatternBlockType.Gap;
                patternBlock.Start = 0;
                patternBlock.Stop = long.MaxValue;
            }
            else
            {
                patternBlock.Type = PatternBlockType.Elements;
                patternBlock.Start = i;
                patternBlock.Stop = i;
            }
        }
        else if (patternBlock.Type == PatternBlockType.Elements)
        {
            if (_patternSequence[i] == _sequences.Constants.Any)
            {
                pattern.Add(patternBlock);
                patternBlock = new PatternBlock
                {
                    Type = PatternBlockType.Gap,
                    Start = 1,
                    Stop = 1
                };
            }
            else if (_patternSequence[i] == ZeroOrMany)
            {
                pattern.Add(patternBlock);
                patternBlock = new PatternBlock
                {
                    Type = PatternBlockType.Gap,
                    Start = 0,
                    Stop = long.MaxValue
                };
            }
            else
            {
                patternBlock.Stop = i;
            }
        }
        else // patternBlock.Type == PatternBlockType.Gap
        {
            if (_patternSequence[i] == _sequences.Constants.Any)
            {
                patternBlock.Start++;
                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;

```

```

1924 //}
1925
1926 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1927 private bool PatternMatchCore(LinkIndex element)
1928 {
1929     if (_patternPosition >= _pattern.Count)
1930     {
1931         _patternPosition = -2;
1932         return false;
1933     }
1934     var currentPatternBlock = _pattern[_patternPosition];
1935     if (currentPatternBlock.Type == PatternBlockType.Gap)
1936     {
1937         //var currentMatchingBlockLength = (_sequencePosition -
1938         ↪ _lastMatchedBlockPosition);
1939         if (_sequencePosition < currentPatternBlock.Start)
1940         {
1941             _sequencePosition++;
1942             return true; // Двигаемся дальше
1943         }
1944         // Это последний блок
1945         if (_pattern.Count == _patternPosition + 1)
1946         {
1947             _patternPosition++;
1948             _sequencePosition = 0;
1949             return false; // Полное соответствие
1950         }
1951         else
1952         {
1953             if (_sequencePosition > currentPatternBlock.Stop)
1954             {
1955                 return false; // Соответствие невозможно
1956             }
1957             var nextPatternBlock = _pattern[_patternPosition + 1];
1958             if (_patternSequence[nextPatternBlock.Start] == element)
1959             {
1960                 if (nextPatternBlock.Start < nextPatternBlock.Stop)
1961                 {
1962                     _patternPosition++;
1963                     _sequencePosition = 1;
1964                 }
1965                 else
1966                 {
1967                     _patternPosition += 2;
1968                     _sequencePosition = 0;
1969                 }
1970             }
1971         }
1972     }
1973     else // currentPatternBlock.Type == PatternBlockType.Elements
1974     {
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     //}
2000     //if (_filterPosition == _patternSequence.Length)
2001     //{
2002         _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 }

```

#### 1.149 ./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,

```



```

34  /// так же проблема может быть решена при реализации дистанционных триггеров.
35  /// Нужны ли уникальные указатели вообще?
36  /// Что если обращение к информации будет происходить через содержимое всегда?
37  ///
38  /// Писать тесты.
39  ///
40  ///
41  /// Можно убрать зависимость от конкретной реализации Links,
42  /// на зависимость от абстрактного элемента, который может быть представлен несколькими
    ↳ 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
175 #endregion
176
177 #region Create
178
179 [MethodImpl(MethodImplOptions.AggressiveInlining)]
180 public LinkIndex Create(ICollection<LinkIndex> restrictions)
181 {
182     return _sync.ExecuteWriteOperation(() =>
183     {
184         if (restrictions.IsNullOrEmpty())
185         {
186             return Constants.Null;
187         }

```

```

187         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
188         return CreateCore(restrictions);
189     });
190 }
191
192 [MethodImpl(MethodImplOptions.AggressiveInlining)]
193 private LinkIndex CreateCore(ICollection<LinkIndex> restrictions)
194 {
195     LinkIndex[] sequence = restrictions.SkipFirst();
196     if (Options.UseIndex)
197     {
198         Options.Index.Add(sequence);
199     }
200     var sequenceRoot = default(LinkIndex);
201     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
202     {
203         var matches = Each(restrictions);
204         if (matches.Count > 0)
205         {
206             sequenceRoot = matches[0];
207         }
208     }
209     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
210     {
211         return CompactCore(sequence);
212     }
213     if (sequenceRoot == default)
214     {
215         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
216     }
217     if (Options.UseSequenceMarker)
218     {
219         return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
220     }
221     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
222 }
223
224 #endregion
225
226 #region Each
227
228 [MethodImpl(MethodImplOptions.AggressiveInlining)]
229 public List<LinkIndex> Each(ICollection<LinkIndex> sequence)
230 {
231     var results = new List<LinkIndex>();
232     var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
233     Each(filler.AddFirstAndReturnConstant, sequence);
234     return results;
235 }
236
237 [MethodImpl(MethodImplOptions.AggressiveInlining)]
238 public LinkIndex Each(Func<ICollection<LinkIndex>, LinkIndex> handler, ICollection<LinkIndex>
    → restrictions)
239 {
240     return _sync.ExecuteReadOperation(() =>
241     {
242         if (restrictions.IsNullOrEmpty())
243         {
244             return Constants.Continue;
245         }
246         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
247         if (restrictions.Count == 1)
248         {
249             var link = restrictions[0];
250             var any = Constants.Any;
251             if (link == any)
252             {
253                 if (Options.UseSequenceMarker)
254                 {
255                     return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
    → Options.SequenceMarkerLink, any));
256                 }
257                 else
258                 {
259                     return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
    → any));
260                 }
261             }

```

```

262         if (Options.UseSequenceMarker)
263         {
264             var sequenceLinkValues = Links.Unsync.GetLink(link);
265             if (sequenceLinkValues[Constants.SourcePart] ==
                ↳ Options.SequenceMarkerLink)
266             {
267                 link = sequenceLinkValues[Constants.TargetPart];
268             }
269         }
270         var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
271         sequence[0] = link;
272         return handler(sequence);
273     }
274     else if (restrictions.Count == 2)
275     {
276         throw new NotImplementedException();
277     }
278     else if (restrictions.Count == 3)
279     {
280         return Links.Unsync.Each(handler, restrictions);
281     }
282     else
283     {
284         var sequence = restrictions.SkipFirst();
285         if (Options.UseIndex && !Options.Index.MightContain(sequence))
286         {
287             return Constants.Break;
288         }
289         return EachCore(handler, sequence);
290     }
291 });
292 }
293
294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
    ↳ values)
296 {
297     var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
298     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
299     ↳ Id.
300     Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
301     ↳ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
302     ↳ matcher.HandleFullMatched;
303     //if (sequence.Length >= 2)
304     if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
305     {
306         return Constants.Break;
307     }
308     var last = values.Count - 2;
309     for (var i = 1; i < last; i++)
310     {
311         if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
            ↳ Constants.Continue)
312         {
313             return Constants.Break;
314         }
315     }
316     if (values.Count >= 3)
317     {
318         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
            ↳ != Constants.Continue)
319         {
320             return Constants.Break;
321         }
322     }
323     return Constants.Continue;
324 }
325
326 [MethodImpl(MethodImplOptions.AggressiveInlining)]
327 private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ left, LinkIndex right)
328 {
329     return Links.Unsync.Each(doublet =>
330     {
331         var doubletIndex = doublet[Constants.IndexPart];
332         if (StepRight(handler, doubletIndex, right) != Constants.Continue)
333         {
334             return Constants.Break;
335         }
336     });
337 }

```

```

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             {

```

```

478         Links.Unsync.MergeAndDelete(sequence, newSequence);
479     }
480 }
481 }
482 }
483
484 #endregion
485
486 #region Delete
487
488 [MethodImpl(MethodImplOptions.AggressiveInlining)]
489 public void Delete(ICollection<LinkIndex> restrictions)
490 {
491     _sync.ExecuteWriteOperation(() =>
492     {
493         var sequence = restrictions.SkipFirst();
494         // TODO: Check all options only ones before loop execution
495         foreach (var linkToDelete in Each(sequence))
496         {
497             DeleteOneCore(linkToDelete);
498         }
499     });
500 }
501
502 [MethodImpl(MethodImplOptions.AggressiveInlining)]
503 private void DeleteOneCore(LinkIndex link)
504 {
505     if (Options.UseGarbageCollection)
506     {
507         var sequenceElements = GetSequenceElements(link);
508         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
509         var sequenceLink = GetSequenceByElements(sequenceElements);
510         if (Options.UseCascadeDelete || CountUsages(link) == 0)
511         {
512             if (sequenceLink != Constants.Null)
513             {
514                 Links.Unsync.Delete(sequenceLink);
515             }
516             Links.Unsync.Delete(link);
517         }
518         ClearGarbage(sequenceElementsContents.Source);
519         ClearGarbage(sequenceElementsContents.Target);
520     }
521     else
522     {
523         if (Options.UseSequenceMarker)
524         {
525             var sequenceElements = GetSequenceElements(link);
526             var sequenceLink = GetSequenceByElements(sequenceElements);
527             if (Options.UseCascadeDelete || CountUsages(link) == 0)
528             {
529                 if (sequenceLink != Constants.Null)
530                 {
531                     Links.Unsync.Delete(sequenceLink);
532                 }
533                 Links.Unsync.Delete(link);
534             }
535         }
536         else
537         {
538             if (Options.UseCascadeDelete || CountUsages(link) == 0)
539             {
540                 Links.Unsync.Delete(link);
541             }
542         }
543     }
544 }
545
546 #endregion
547
548 #region Compactification
549
550 [MethodImpl(MethodImplOptions.AggressiveInlining)]
551 public void CompactAll()
552 {
553     _sync.ExecuteWriteOperation(() =>
554     {
555         var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
556         for (int i = 0; i < sequences.Count; i++)

```

```

557         {
558             var sequence = this.ToList(sequences[i]);
559             Compact(sequence.ShiftRight());
560         }
561     });
562 }
563
564 /// <remarks>
565 /// bestVariant можно выбирать по максимальному числу использований,
566 /// но балансированный позволяет гарантировать уникальность (если есть возможность,
567 /// гарантировать его использование в других местах).
568 ///
569 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
570 /// </remarks>
571 [MethodImpl(MethodImplOptions.AggressiveInlining)]
572 public LinkIndex Compact(ICollection<LinkIndex> sequence)
573 {
574     return _sync.ExecuteWriteOperation(() =>
575     {
576         if (sequence.IsNullOrEmpty())
577         {
578             return Constants.Null;
579         }
580         Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
581         return CompactCore(sequence);
582     });
583 }
584
585 [MethodImpl(MethodImplOptions.AggressiveInlining)]
586 private LinkIndex CompactCore(ICollection<LinkIndex> sequence) => UpdateCore(sequence,
587     ↪ sequence);
588
589 #endregion
590
591 #region Garbage Collection
592
593 /// <remarks>
594 /// 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 {

```



```

633 private readonly Sequences _sequences;
634 private readonly IList<LinkIndex> _patternSequence;
635 private readonly HashSet<LinkIndex> _linksInSequence;
636 private readonly HashSet<LinkIndex> _results;
637 private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
638 private readonly HashSet<LinkIndex> _readAsElements;
639 private int _filterPosition;
640
641 [MethodImpl(MethodImplOptions.AggressiveInlining)]
642 public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
    ↳ HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
    ↳ HashSet<LinkIndex> readAsElements = null)
    : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
643 {
644     _sequences = sequences;
645     _patternSequence = patternSequence;
646     _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
    ↳ _links.Constants.Any && x != ZeroOrMany));
647     _results = results;
648     _stopableHandler = stopableHandler;
649     _readAsElements = readAsElements;
650 }
651
652 [MethodImpl(MethodImplOptions.AggressiveInlining)]
653 protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
654     ↳ (_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
679         && element != _patternSequence[_filterPosition])
680     {
681         _filterPosition = -1;
682         return false; // Начинается/Продолжается иначе
683     }
684     _filterPosition++;
685     return true;
686 }
687
688 [MethodImpl(MethodImplOptions.AggressiveInlining)]
689 public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
690 {
691     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
692     if (FullMatch(sequenceToMatch))
693     {
694         _results.Add(sequenceToMatch);
695     }
696 }
697
698 [MethodImpl(MethodImplOptions.AggressiveInlining)]
699 public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
700 {
701     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
702     if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
703     {
704         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
705     }
706     return _links.Constants.Continue;
707 }

```

```

708 [MethodImpl(MethodImplOptions.AggressiveInlining)]
709 public LinkIndex HandleFullMatchedSequence(ICollection<LinkIndex> restrictions)
710 {
711     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
712     var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
713     if (sequence != _links.Constants.Null && FullMatch(sequenceToMatch) &&
714         ↪ _results.Add(sequenceToMatch))
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 }
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 }

```

#### 1.150 ./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 }

```

#### 1.151 ./csharp/Platform.Data.Doublets/Sequences/SequencesOptions.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Collections.Stacks;
5  using Platform.Converters;
6  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
7  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
8  using Platform.Data.Doublets.Sequences.Converters;
9  using Platform.Data.Doublets.Sequences.Walkers;
10 using Platform.Data.Doublets.Sequences.Indexes;

```

```

11 using Platform.Data.Doublets.Sequences.CriterionMatchers;
12 using System.Runtime.CompilerServices;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
        ↳ ILinks<TLink> must contain GetConstants function.
19     {
20         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
21
22         public TLink SequenceMarkerLink
23         {
24             [MethodImpl(MethodImplOptions.AggressiveInlining)]
25             get;
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             set;
28         }
29
30         public bool UseCascadeUpdate
31         {
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             get;
34             [MethodImpl(MethodImplOptions.AggressiveInlining)]
35             set;
36         }
37
38         public bool UseCascadeDelete
39         {
40             [MethodImpl(MethodImplOptions.AggressiveInlining)]
41             get;
42             [MethodImpl(MethodImplOptions.AggressiveInlining)]
43             set;
44         }
45
46         public bool UseIndex
47         {
48             [MethodImpl(MethodImplOptions.AggressiveInlining)]
49             get;
50             [MethodImpl(MethodImplOptions.AggressiveInlining)]
51             set;
52         } // TODO: Update Index on sequence update/delete.
53
54         public bool UseSequenceMarker
55         {
56             [MethodImpl(MethodImplOptions.AggressiveInlining)]
57             get;
58             [MethodImpl(MethodImplOptions.AggressiveInlining)]
59             set;
60         }
61
62         public bool UseCompression
63         {
64             [MethodImpl(MethodImplOptions.AggressiveInlining)]
65             get;
66             [MethodImpl(MethodImplOptions.AggressiveInlining)]
67             set;
68         }
69
70         public bool UseGarbageCollection
71         {
72             [MethodImpl(MethodImplOptions.AggressiveInlining)]
73             get;
74             [MethodImpl(MethodImplOptions.AggressiveInlining)]
75             set;
76         }
77
78         public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting
79         {
80             [MethodImpl(MethodImplOptions.AggressiveInlining)]
81             get;
82             [MethodImpl(MethodImplOptions.AggressiveInlining)]
83             set;
84         }
85
86         public bool EnforceSingleSequenceVersionOnWriteBasedOnNew
87         {
88             [MethodImpl(MethodImplOptions.AggressiveInlining)]
89             get;

```

```

90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     set;
92 }
93
94 public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher
95 {
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     get;
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     set;
100 }
101
102 public IConverter<IList<TLink>, TLink> LinksToSequenceConverter
103 {
104     [MethodImpl(MethodImplOptions.AggressiveInlining)]
105     get;
106     [MethodImpl(MethodImplOptions.AggressiveInlining)]
107     set;
108 }
109
110 public ISequenceIndex<TLink> Index
111 {
112     [MethodImpl(MethodImplOptions.AggressiveInlining)]
113     get;
114     [MethodImpl(MethodImplOptions.AggressiveInlining)]
115     set;
116 }
117
118 public ISequenceWalker<TLink> Walker
119 {
120     [MethodImpl(MethodImplOptions.AggressiveInlining)]
121     get;
122     [MethodImpl(MethodImplOptions.AggressiveInlining)]
123     set;
124 }
125
126 public bool ReadFullSequence
127 {
128     [MethodImpl(MethodImplOptions.AggressiveInlining)]
129     get;
130     [MethodImpl(MethodImplOptions.AggressiveInlining)]
131     set;
132 }
133
134 // TODO: Реализовать компактификацию при чтении
135 //public bool EnforceSingleSequenceVersionOnRead { get; set; }
136 //public bool UseRequestMarker { get; set; }
137 //public bool StoreRequestResults { get; set; }
138
139 [MethodImpl(MethodImplOptions.AggressiveInlining)]
140 public void InitOptions(ISynchronizedLinks<TLink> links)
141 {
142     if (UseSequenceMarker)
143     {
144         if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
145         {
146             SequenceMarkerLink = links.CreatePoint();
147         }
148         else
149         {
150             if (!links.Exists(SequenceMarkerLink))
151             {
152                 var link = links.CreatePoint();
153                 if (!_equalityComparer.Equals(link, SequenceMarkerLink))
154                 {
155                     throw new InvalidOperationException("Cannot recreate sequence marker
156                                     ↪ link.");
157                 }
158             }
159             if (MarkedSequenceMatcher == null)
160             {
161                 MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
162                                     ↪ SequenceMarkerLink);
163             }
164             var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
165             if (UseCompression)
166             {

```

```

167         if (LinksToSequenceConverter == null)
168         {
169             ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
170             if (UseSequenceMarker)
171             {
172                 totalSequenceSymbolFrequencyCounter = new
173                     ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
174                     ↪ MarkedSequenceMatcher);
175             }
176             else
177             {
178                 totalSequenceSymbolFrequencyCounter = new
179                     ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
180             }
181             var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
182                 ↪ totalSequenceSymbolFrequencyCounter);
183             var compressingConverter = new CompressingConverter<TLink>(links,
184                 ↪ balancedVariantConverter, doubletFrequenciesCache);
185             LinksToSequenceConverter = compressingConverter;
186         }
187     }
188     else
189     {
190         if (LinksToSequenceConverter == null)
191         {
192             LinksToSequenceConverter = balancedVariantConverter;
193         }
194         if (UseIndex && Index == null)
195         {
196             Index = new SequenceIndex<TLink>(links);
197         }
198         if (Walker == null)
199         {
200             Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
201         }
202     }
203 }
204
205 [MethodImpl(MethodImplOptions.AggressiveInlining)]
206 public void ValidateOptions()
207 {
208     if (UseGarbageCollection && !UseSequenceMarker)
209     {
210         throw new NotSupportedException("To use garbage collection UseSequenceMarker
211             ↪ option must be on.");
212     }
213 }

```

#### 1.152 ./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 }

```

#### 1.153 ./csharp/Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Collections.Stacks;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Sequences.Walkers
9 {
10     public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

13     public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
14         ↪ isElement) : base(links, stack, isElement) { }
15
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
18         ↪ links.IsPartialPoint) { }
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     protected override TLink GetNextElementAfterPop(TLink element) =>
22         ↪ _links.GetSource(element);
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     protected override TLink GetNextElementAfterPush(TLink element) =>
26         ↪ _links.GetTarget(element);
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     protected override IEnumerable<TLink> WalkContents(TLink element)
30     {
31         var links = _links;
32         var parts = links.GetLink(element);
33         var start = links.Constants.SourcePart;
34         for (var i = parts.Count - 1; i >= start; i--)
35         {
36             var part = parts[i];
37             if (IsElement(part))
38             {
39                 yield return part;
40             }
41         }
42     }
43 }

```

#### 1.154 ./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             } while (hasElements = false);
47         }
48     }
49 #if USEARRAYPOOL

```

```

44         var nextArray = ArrayPool.Allocate<ulong>(length);
45     #else
46         var nextArray = new TLink[length];
47     #endif
48     hasElements = false;
49     for (var i = 0; i < array.Length; i++)
50     {
51         var candidate = array[i];
52         if (!_equalityComparer.Equals(array[i], default))
53         {
54             continue;
55         }
56         var doubletOffset = i * 2;
57         if (!_isElement(candidate))
58         {
59             nextArray[doubletOffset] = candidate;
60         }
61         else
62         {
63             var links = _links;
64             var link = links.GetLink(candidate);
65             var linkSource = links.GetSource(link);
66             var linkTarget = links.GetTarget(link);
67             nextArray[doubletOffset] = linkSource;
68             nextArray[doubletOffset + 1] = linkTarget;
69             if (!hasElements)
70             {
71                 hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
72             }
73         }
74     }
75     #if USEARRAYPOOL
76     if (array.Length > 1)
77     {
78         ArrayPool.Free(array);
79     }
80     #endif
81     array = nextArray;
82 }
83 while (hasElements);
84 var filledElementsCount = CountFilledElements(array);
85 if (filledElementsCount == array.Length)
86 {
87     return array;
88 }
89 else
90 {
91     return CopyFilledElements(array, filledElementsCount);
92 }
93 }
94
95 [MethodImpl(MethodImplOptions.AggressiveInlining)]
96 private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
97 {
98     var finalArray = new TLink[filledElementsCount];
99     for (int i = 0, j = 0; i < array.Length; i++)
100     {
101         if (!_equalityComparer.Equals(array[i], default))
102         {
103             finalArray[j] = array[i];
104             j++;
105         }
106     }
107     #if USEARRAYPOOL
108     ArrayPool.Free(array);
109     #endif
110     return finalArray;
111 }
112
113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
114 private static int CountFilledElements(TLink[] array)
115 {
116     var count = 0;
117     for (var i = 0; i < array.Length; i++)
118     {
119         if (!_equalityComparer.Equals(array[i], default))
120         {
121             count++;
122         }
123     }

```



```

123     }
124     return count;
125 }
126 }
127 }

```

### 1.155 ./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 }

```

### 1.156 ./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)]

```

```

26 public IEnumerable<TLink> Walk(TLink sequence)
27 {
28     _stack.Clear();
29     var element = sequence;
30     if (IsElement(element))
31     {
32         yield return element;
33     }
34     else
35     {
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 }

```

### 1.157 ./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)]

```

```

31     public TLink Peek() => _links.GetTarget(GetTop());
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     public TLink Pop()
35     {
36         var element = Peek();
37         if (!_equalityComparer.Equals(element, _stack))
38         {
39             var top = GetTop();
40             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 }

```

### 1.158 ./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 }

```

### 1.159 ./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
        ↳ ReaderWriterLockSynchronization(), links) { }
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
47     {
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 }

```

#### 1.160 ./csharp/Platform.Data.Doublets/Time/DateTimeToLongRawNumberSequenceConverter.cs

```

1  using System;
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.Time
8  {
9      public class DateTimeToLongRawNumberSequenceConverter<TLink> : IConverter<DateTime, TLink>
10     {
11         private readonly IConverter<long, TLink> _int64ToLongRawNumberConverter;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public DateTimeToLongRawNumberSequenceConverter(IConverter<long, TLink>
            ↳ int64ToLongRawNumberConverter) => _int64ToLongRawNumberConverter =
            ↳ int64ToLongRawNumberConverter;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public TLink Convert(DateTime source) =>
            ↳ _int64ToLongRawNumberConverter.Convert(source.ToFileTimeUtc());
18     }
19 }

```

#### 1.161 ./csharp/Platform.Data.Doublets/Time/LongRawNumberSequenceToDateTimeConverter.cs

```

1  using System;
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.Time
8 {
9     public class LongRawNumberSequenceToDateTimeConverter<TLink> : IConverter<TLink, DateTime>
10     {
11         private readonly IConverter<TLink, long> _longRawNumberConverterToInt64;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public LongRawNumberSequenceToDateTimeConverter(IConverter<TLink, long>
15             ↪ longRawNumberConverterToInt64) => _longRawNumberConverterToInt64 =
16             ↪ longRawNumberConverterToInt64;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public DateTime Convert(TLink source) =>
20             ↪ DateTime.FromFileTimeUtc(_longRawNumberConverterToInt64.Convert(source));
21     }
22 }

```

## 1.162 ./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs

```

1 using System;
2 using System.Text;
3 using System.Collections.Generic;
4 using System.Runtime.CompilerServices;
5 using Platform.Singletons;
6 using Platform.Data.Doublets.Unicode;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets
11 {
12     public static class UInt64LinksExtensions
13     {
14         public static readonly LinksConstants<ulong> Constants =
15             ↪ Default<LinksConstants<ulong>>.Instance;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
22         {
23             if (sequence == null)
24             {
25                 return false;
26             }
27             var constants = links.Constants;
28             for (var i = 0; i < sequence.Length; i++)
29             {
30                 if (sequence[i] == constants.Any)
31                 {
32                     return true;
33                 }
34             }
35             return false;
36         }
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
40             ↪ Func<Link<ulong>, bool> isElement, bool renderIndex = false, bool renderDebug =
41             ↪ false)
42         {
43             var sb = new StringBuilder();
44             var visited = new HashSet<ulong>();
45             links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
46                 ↪ innerSb.Append(link.Index), renderIndex, renderDebug);
47             return sb.ToString();
48         }
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
52             ↪ Func<Link<ulong>, bool> isElement, Action<StringBuilder, Link<ulong>> appendElement,
53             ↪ bool renderIndex = false, bool renderDebug = false)
54         {
55             var sb = new StringBuilder();
56             var visited = new HashSet<ulong>();
57             links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
58                 ↪ renderDebug);
59         }
60     }
61 }

```

```

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

```

### 1.163 ./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

```

```

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();
125
126         [MethodImpl(MethodImplOptions.AggressiveInlining)]
127         public bool Equals(Transition other) => TransactionId == other.TransactionId &&
128             ↪ Before == other.Before && After == other.After && Timestamp == other.Timestamp;
129
130         [MethodImpl(MethodImplOptions.AggressiveInlining)]
131         public static bool operator ==(Transition left, Transition right) =>
132             ↪ left.Equals(right);
133
134         [MethodImpl(MethodImplOptions.AggressiveInlining)]
135         public static bool operator !=(Transition left, Transition right) => !(left ==
136             ↪ right);
137     }
138
139     /// <remarks>
140     /// Другие варианты реализации транзакций (атомарности):
141     /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
142     /// ↪ Target)) и индексов.
143     /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
144     /// ↪ потребуется решить вопрос
145     /// со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
146     /// ↪ пересечениями идентификаторов.
147     ///

```



```

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)
    ↪ будут записаны в лог.
154 ///
155 ///
156 </remarks>
157 public class Transaction : DisposableBase
158 {
159     private readonly Queue<Transition> _transitions;
160     private readonly UInt64LinksTransactionsLayer _layer;
161     public bool IsCommitted { get; private set; }
162     public bool IsReverted { get; private set; }
163
164     [MethodImpl(MethodImplOptions.AggressiveInlining)]
165     public Transaction(UInt64LinksTransactionsLayer layer)
166     {
167         _layer = layer;
168         if (_layer._currentTransactionId != 0)
169         {
170             throw new NotSupportedException("Nested transactions not supported.");
171         }
172         IsCommitted = false;
173         IsReverted = false;
174         _transitions = new Queue<Transition>();
175         SetCurrentTransaction(layer, this);
176     }
177
178     [MethodImpl(MethodImplOptions.AggressiveInlining)]
179     public void Commit()
180     {
181         EnsureTransactionAllowsWriteOperations(this);
182         while (_transitions.Count > 0)
183         {
184             var transition = _transitions.Dequeue();
185             _layer._transitions.Enqueue(transition);
186         }
187         _layer._lastCommittedTransactionId = _layer._currentTransactionId;
188         IsCommitted = true;
189     }
190
191     [MethodImpl(MethodImplOptions.AggressiveInlining)]
192     private void Revert()
193     {
194         EnsureTransactionAllowsWriteOperations(this);
195         var transitionsToRevert = new Transition[_transitions.Count];
196         _transitions.CopyTo(transitionsToRevert, 0);
197         for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
198         {
199             _layer.RevertTransition(transitionsToRevert[i]);
200         }
201         IsReverted = true;
202     }
203
204     [MethodImpl(MethodImplOptions.AggressiveInlining)]
205     public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
    ↪ Transaction transaction)
206     {
207         layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
208         layer._currentTransactionTransitions = transaction._transitions;

```

```

209         layer._currentTransaction = transaction;
210     }
211
212     [MethodImpl(MethodImplOptions.AggressiveInlining)]
213     public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
214     {
215         if (transaction.IsReverted)
216         {
217             throw new InvalidOperationException("Transation is reverted.");
218         }
219         if (transaction.IsCommitted)
220         {
221             throw new InvalidOperationException("Transation is committed.");
222         }
223     }
224
225     [MethodImpl(MethodImplOptions.AggressiveInlining)]
226     protected override void Dispose(bool manual, bool wasDisposed)
227     {
228         if (!wasDisposed && _layer != null && !_layer.Disposable.IsDisposed)
229         {
230             if (!IsCommitted && !IsReverted)
231             {
232                 Revert();
233             }
234             _layer.ResetCurrentTransation();
235         }
236     }
237 }
238
239 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
240
241 private readonly string _logAddress;
242 private readonly FileStream _log;
243 private readonly Queue<Transition> _transitions;
244 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
245 private Task _transitionsPusher;
246 private Transition _lastCommitedTransition;
247 private ulong _currentTransactionId;
248 private Queue<Transition> _currentTransactionTransitions;
249 private Transaction _currentTransaction;
250 private ulong _lastCommittedTransactionId;
251
252 [MethodImpl(MethodImplOptions.AggressiveInlining)]
253 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
254     : base(links)
255 {
256     if (string.IsNullOrEmpty(logAddress))
257     {
258         throw new ArgumentNullException(nameof(logAddress));
259     }
260     // В первой строке файла хранится последняя закоммиченную транзакцию.
261     // При запуске это используется для проверки удачного закрытия файла лога.
262     // In the first line of the file the last committed transaction is stored.
263     // On startup, this is used to check that the log file is successfully closed.
264     var lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
265     var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
266     if (!lastCommittedTransition.Equals(lastWrittenTransition))
267     {
268         Dispose();
269         throw new NotSupportedException("Database is damaged, autorecovery is not
270             ↳ supported yet.");
271     }
272     if (lastCommittedTransition == default)
273     {
274         FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
275     }
276     _lastCommittedTransition = lastCommittedTransition;
277     // TODO: Think about a better way to calculate or store this value
278     var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
279     _lastCommittedTransactionId = allTransitions.Length > 0 ? allTransitions.Max(x =>
280         ↳ x.TransactionId) : 0;
281     _uniqueTimestampFactory = new UniqueTimestampFactory();
282     _logAddress = logAddress;
283     _log = FileHelpers.Append(logAddress);
284     _transitions = new Queue<Transition>();
285     _transitionsPusher = new Task(TransitionsPusher);
286     _transitionsPusher.Start();
287 }

```

```

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,
296         ↪ default, createdLink));
297     return createdLinkIndex;
298 }
299
300 [MethodImpl(MethodImplOptions.AggressiveInlining)]
301 public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
302 {
303     var linkIndex = restrictions[_constants.IndexPart];
304     var beforeLink = new Link<ulong>(_links.GetLink(linkIndex));
305     linkIndex = _links.Update(restrictions, substitution);
306     var afterLink = new Link<ulong>(_links.GetLink(linkIndex));
307     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
308         ↪ beforeLink, afterLink));
309     return linkIndex;
310 }
311
312 [MethodImpl(MethodImplOptions.AggressiveInlining)]
313 public override void Delete(IList<ulong> restrictions)
314 {
315     var link = restrictions[_constants.IndexPart];
316     var deletedLink = new Link<ulong>(_links.GetLink(link));
317     _links.Delete(link);
318     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
319         ↪ deletedLink, default));
320 }
321
322 [MethodImpl(MethodImplOptions.AggressiveInlining)]
323 private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
324     ↪ _transitions;
325
326 [MethodImpl(MethodImplOptions.AggressiveInlining)]
327 private void CommitTransition(Transition transition)
328 {
329     if (_currentTransaction != null)
330     {
331         Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
332     }
333     var transitions = GetCurrentTransitions();
334     transitions.Enqueue(transition);
335 }
336
337 [MethodImpl(MethodImplOptions.AggressiveInlining)]
338 private void RevertTransition(Transition transition)
339 {
340     if (transition.After.IsNull()) // Revert Deletion with Creation
341     {
342         _links.Create();
343     }
344     else if (transition.Before.IsNull()) // Revert Creation with Deletion
345     {
346         _links.Delete(transition.After.Index);
347     }
348     else // Revert Update
349     {
350         _links.Update(new[] { transition.After.Index, transition.Before.Source,
351             ↪ transition.Before.Target });
352     }
353 }
354
355 [MethodImpl(MethodImplOptions.AggressiveInlining)]
356 private void ResetCurrentTransation()
357 {
358     _currentTransactionId = 0;
359     _currentTransactionTransitions = null;
360     _currentTransaction = null;
361 }
362
363 [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 }

```

#### 1.164 ./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     {

```

```

10     private static readonly UncheckedConverter<char, TLink> _charToAddressConverter =
11         ↳ UncheckedConverter<char, TLink>.Default;
12
13     private readonly IConverter<TLink> _addressToNumberConverter;
14     private readonly TLink _unicodeSymbolMarker;
15
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
18         ↳ addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
19     {
20         _addressToNumberConverter = addressToNumberConverter;
21         _unicodeSymbolMarker = unicodeSymbolMarker;
22     }
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     public TLink Convert(char source)
26     {
27         var unaryNumber =
28             ↳ _addressToNumberConverter.Convert(_charToAddressConverter.Convert(source));
29         return _links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
30     }
31 }

```

### 1.165 ./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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
36             ↳ charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
37             ↳ TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker)
38             : this(links, new
39                 ↳ StringToUnicodeSymbolsListConverter<TLink>(charToUnicodeSymbolConverter), index,
40                 ↳ listToSequenceLinkConverter, unicodeSequenceMarker) { }
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
44             ↳ charToUnicodeSymbolConverter, IConverter<IList<TLink>, TLink>
45             ↳ listToSequenceLinkConverter, TLink unicodeSequenceMarker)
46             : this(links, charToUnicodeSymbolConverter, new Unindex<TLink>(),
47                 ↳ listToSequenceLinkConverter, unicodeSequenceMarker) { }
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
51             ↳ IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
52             ↳ listToSequenceLinkConverter, TLink unicodeSequenceMarker)
53             : this(links, stringToUnicodeSymbolListConverter, new Unindex<TLink>(),
54                 ↳ listToSequenceLinkConverter, unicodeSequenceMarker) { }
55     }
56 }

```

```

37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     public TLink Convert(string source)
39     {
40         var elements = _stringToUnicodeSymbolListConverter.Convert(source);
41         return _unicodeSymbolListToSequenceConverter.Convert(elements);
42     }
43 }
44 }
45 }

```

#### 1.166 ./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>
15             ↪ charToUnicodeSymbolConverter) => _charToUnicodeSymbolConverter =
16             ↪ charToUnicodeSymbolConverter;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public IList<TLink> Convert(string source)
20         {
21             var elements = new TLink[source.Length];
22             for (var i = 0; i < elements.Length; i++)
23             {
24                 elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
25             }
26             return elements;
27         }
28     }
29 }

```

#### 1.167 ./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         }
40     }
41 }

```

```

39     _initialized = true;
40     var firstLink = _links.CreatePoint();
41     if (firstLink != FirstCharLink)
42     {
43         _links.Delete(firstLink);
44     }
45     else
46     {
47         for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
48         {
49             // From NIL to It (NIL -> Character) transformation meaning, (or infinite
50             // ↪ amount of NIL characters before actual Character)
51             var createdLink = _links.CreatePoint();
52             _links.Update(createdLink, firstLink, createdLink);
53             if (createdLink != i)
54             {
55                 throw new InvalidOperationException("Unable to initialize UTF 16
56                 ↪ table.");
57             }
58         }
59     }
60     // 0 - null link
61     // 1 - nil character (0 character)
62     // ...
63     // 65536 (0(1) + 65535 = 65536 possible values)
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     public static ulong FromCharToLink(char character) => (ulong)character + 1;
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     public static char FromLinkToChar(ulong link) => (char)(link - 1);
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     public static bool IsCharLink(ulong link) => link <= MapSize;
73
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     public static string FromLinksToString(IList<ulong> linksList)
76     {
77         var sb = new StringBuilder();
78         for (int i = 0; i < linksList.Count; i++)
79         {
80             sb.Append(FromLinkToChar(linksList[i]));
81         }
82         return sb.ToString();
83     }
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
87     {
88         var sb = new StringBuilder();
89         if (links.Exists(link))
90         {
91             StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
92             x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
93             ↪ element =>
94             {
95                 sb.Append(FromLinkToChar(element));
96                 return true;
97             });
98         }
99         return sb.ToString();
100     }
101
102     [MethodImpl(MethodImplOptions.AggressiveInlining)]
103     public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
104     ↪ chars.Length);
105
106     [MethodImpl(MethodImplOptions.AggressiveInlining)]
107     public static ulong[] FromCharsToLinkArray(char[] chars, int count)
108     {
109         // char array to ulong array
110         var linksSequence = new ulong[count];
111         for (var i = 0; i < count; i++)
112         {
113             linksSequence[i] = FromCharToLink(chars[i]);
114         }
115     }

```

```

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

```



```

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

```

#### 1.168 ./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             [MethodImpl(MethodImplOptions.AggressiveInlining)]
29             public string Convert(TLink source)
30             {
31                 if (!_unicodeSequenceCriterionMatcher.IsMatched(source))
32                 {
33                     throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
34                         ↳ not a unicode sequence.");
35                 }
36                 var sequence = _links.GetSource(source);
37                 var sb = new StringBuilder();
38                 foreach (var character in _sequenceWalker.Walk(sequence))
39                 {
40                     sb.Append(_unicodeSymbolToCharConverter.Convert(character));
41                 }
42                 return sb.ToString();
43             }
44         }
45     }
46 }

```

#### 1.169 ./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)]

```

```

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

```

#### 1.170 ./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>,
        ↳ IConverter<IList<TLink>, TLink>
11     {
12         private readonly ISequenceIndex<TLink> _index;
13         private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
14         private readonly TLink _unicodeSequenceMarker;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
        ↳ ISequenceIndex<TLink> index, IConverter<IList<TLink>, TLink>
        ↳ listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
18         {
19             _index = index;
20             _listToSequenceLinkConverter = listToSequenceLinkConverter;
21             _unicodeSequenceMarker = unicodeSequenceMarker;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
        ↳ IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
        ↳ unicodeSequenceMarker)
26             : this(links, new Unindex<TLink>(), listToSequenceLinkConverter,
        ↳ unicodeSequenceMarker) { }
27
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 }

```

#### 1.171 ./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 }

```

#### 1.172 ./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 }

```

#### 1.173 ./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         }
16     }
17 }

```

```

14         var maximum = new Hybrid<ulong>(long.MaxValue, isExternal: true);
15
16         Assert.True(constants.IsExternalReference(minimum));
17         Assert.True(constants.IsExternalReference(maximum));
18     }
19 }
20 }

```

#### 1.174 ./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         Gravida 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.
51         Eget nunc scelerisque viverra mauris in aliquam sem fringilla.
52         Pharetra vel turpis nunc eget lorem dolor sed viverra.
53         Mattis pellentesque id nibh tortor id aliquet.
54         Purus non enim praesent elementum facilisis leo vel.
55         Etiam sit amet nisl purus in mollis nunc sed.
56         Tortor at auctor urna nunc id cursus metus aliquam.
57         Volutpat odio facilisis mauris sit amet.
58         Turpis egestas pretium aenean pharetra magna ac placerat.
59         Fermentum dui faucibus in ornare quam viverra orci sagittis eu.
60         Porttitor leo a diam sollicitudin tempor id eu.
61         Volutpat sed cras ornare arcu dui.
62         Ut aliquam purus sit amet luctus venenatis lectus magna.
63         Aliquet risus feugiat in ante metus dictum at.
64         Mattis nunc sed blandit libero.
65         Elit pellentesque habitant morbi tristique senectus et netus.
66         Nibh sit amet commodo nulla facilisi nullam vehicula ipsum a.
67         Enim sit amet venenatis urna cursus eget nunc scelerisque viverra.
68         Amet venenatis urna cursus eget nunc scelerisque viverra mauris in.
69         Diam donec adipiscing tristique risus nec feugiat.
70         Pulvinar mattis nunc sed blandit libero volutpat.
71         Cras fermentum odio eu feugiat pretium nibh ipsum.
72         In nulla posuere sollicitudin aliquam ultrices sagittis orci a.
73         Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.
74         A iaculis atherat pellentesque.
75         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;
128
129         links.UseUnicode();
130
131         var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
132
133         var totalSequenceSymbolFrequencyCounter = new
134             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
135
136         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
137             ↪ totalSequenceSymbolFrequencyCounter);
138
139         var index = new
140             ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
141         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
142             ↪ ncyNumberConverter<ulong>(linkFrequenciesCache);
143
144         var sequenceToItsLocalElementLevelsConverter = new
145             ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
146             ↪ linkToItsFrequencyNumberConverter);
147         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
148             ↪ sequenceToItsLocalElementLevelsConverter);
149
150         var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
151             ↪ Walker = new LeveledSequenceWalker<ulong>(links) });

```

```

132         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
133             ↪ index, optimalVariantConverter);
134     }
135 }
136
137 private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
138     ↪ SequenceToItsLocalElementLevelsConverter<ulong>
139     ↪ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
140     ↪ OptimalVariantConverter<ulong> optimalVariantConverter)
141 {
142     index.Add(sequence);
143
144     var optimalVariant = optimalVariantConverter.Convert(sequence);
145
146     var readSequence1 = sequences.ToList(optimalVariant);
147
148     Assert.True(sequence.SequenceEqual(readSequence1));
149 }
150
151 [Fact]
152 public static void SavedSequencesOptimizationTest()
153 {
154     LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
155         ↪ (long.MaxValue + 1UL, ulong.MaxValue));
156
157     using (var memory = new HeapResizableDirectMemory())
158     using (var disposableLinks = new UInt64UnitedMemoryLinks(memory,
159         ↪ UInt64UnitedMemoryLinks.DefaultLinksSizeStep, constants, IndexTreeType.Default))
160     {
161         var links = new UInt64Links(disposableLinks);
162
163         var root = links.CreatePoint();
164
165         //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
166         var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
167
168         var unicodeSymbolMarker = links.GetOrCreate(root,
169             ↪ addressToNumberConverter.Convert(1));
170         var unicodeSequenceMarker = links.GetOrCreate(root,
171             ↪ addressToNumberConverter.Convert(2));
172
173         var totalSequenceSymbolFrequencyCounter = new
174             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
175         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
176             ↪ totalSequenceSymbolFrequencyCounter);
177         var index = new
178             ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
179         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
180             ↪ ncyNumberConverter<ulong>(linkFrequenciesCache);
181         var sequenceToItsLocalElementLevelsConverter = new
182             ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
183             ↪ linkToItsFrequencyNumberConverter);
184         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
185             ↪ sequenceToItsLocalElementLevelsConverter);
186
187         var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
188             ↪ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
189
190         var unicodeSequencesOptions = new SequencesOptions<ulong>()
191         {
192             UseSequenceMarker = true,
193             SequenceMarkerLink = unicodeSequenceMarker,
194             UseIndex = true,
195             Index = index,
196             LinksToSequenceConverter = optimalVariantConverter,
197             Walker = walker,
198             UseGarbageCollection = true
199         };
200
201         var unicodeSequences = new Sequences.Sequences(new
202             ↪ SynchronizedLinks<ulong>(links), unicodeSequencesOptions);
203
204         // Create some sequences
205         var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
206             ↪ StringSplitOptions.RemoveEmptyEntries);
207         var arrays = strings.Select(x => x.Select(y =>
208             ↪ addressToNumberConverter.Convert(y)).ToArray()).ToArray();

```

```

191     for (int i = 0; i < arrays.Length; i++)
192     {
193         unicodeSequences.Create(arrays[i].ShiftRight());
194     }
195
196     var linksCountAfterCreation = links.Count();
197
198     // get list of sequences links
199     // for each sequence link
200     //     create new sequence version
201     //     if new sequence is not the same as sequence link
202     //         delete sequence link
203     //         collect garbadge
204     unicodeSequences.CompactAll();
205
206     var linksCountAfterCompactification = links.Count();
207
208     Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
209 }
210 }
211 }
212 }

```

### 1.175 ./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

```

```

55         for (var i = 0; i < sequenceLength; i++)
56         {
57             links.Delete(sequence[i]);
58         }
59     }
60 }
61 }
62 }
63 }

```

# 1.176 ./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
25         [Fact]
26         public static void BasicHeapMemoryTest()
27         {
28             using (var memory = new
29                 ↪ HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
30             using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
31                 ↪ UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
32             {
33                 memoryAdapter.TestBasicMemoryOperations();
34             }
35
36             private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
37             {
38                 var link = memoryAdapter.Create();
39                 memoryAdapter.Delete(link);
40             }
41
42             [Fact]
43             public static void NonexistentReferencesHeapMemoryTest()
44             {
45                 using (var memory = new
46                     ↪ HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
47                 using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
48                     ↪ UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
49                 {
50                     memoryAdapter.TestNonexistentReferences();
51                 }
52
53                 private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
54                 {
55                     var link = memoryAdapter.Create();
56                     memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
57                     var resultLink = _constants.Null;
58                     memoryAdapter.Each(foundLink =>
59                     {
60                         resultLink = foundLink[_constants.IndexPart];
61                         return _constants.Break;
62                     }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
63                     Assert.True(resultLink == link);
64                     Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
65                     memoryAdapter.Delete(link);
66                 }
67             }
68         }
69     }
70 }

```



```
64     }
65 }
```

### 1.177 ./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(Skip = "Would be fixed later.")]
37         public static void FullAutoResolutionTest()
38         {
39             using (var scope = new Scope(autoInclude: true, autoExplore: true))
40             {
41                 var instance = scope.Use<UInt64Links>();
42                 Assert.IsType<UInt64Links>(instance);
43             }
44         }
45
46         [Fact]
47         public static void TypeParametersTest()
48         {
49             using (var scope = new Scope<Types<HeapResizableDirectMemory,
50 ↵      UnitedMemoryLinks<ulong>>>())
51             {
52                 var links = scope.Use<ILinks<ulong>>();
53                 Assert.IsType<UnitedMemoryLinks<ulong>>(links);
54             }
55         }
56     }
```

### 1.178 ./csharp/Platform.Data.Doublets.Tests/SequencesTests.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Linq;
5  using Xunit;
6  using Platform.Collections;
7  using Platform.Collections.Arrays;
8  using Platform.Random;
9  using Platform.IO;
10 using Platform.Singletons;
11 using Platform.Data.Doublets.Sequences;
12 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
13 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
14 using Platform.Data.Doublets.Sequences.Converters;
15 using Platform.Data.Doublets.Unicode;
16
17 namespace Platform.Data.Doublets.Tests
```

```

18 {
19     public static class SequencesTests
20     {
21         private static readonly LinksConstants<ulong> _constants =
22             ↳ Default<LinksConstants<ulong>>.Instance;
23
24         static SequencesTests()
25         {
26             // Trigger static constructor to not mess with performance measurements
27             _ = BitString.GetBitMaskFromIndex(1);
28         }
29
30         [Fact]
31         public static void CreateAllVariantsTest()
32         {
33             const long sequenceLength = 8;
34
35             using (var scope = new TempLinksTestScope(useSequences: true))
36             {
37                 var links = scope.Links;
38                 var sequences = scope.Sequences;
39
40                 var sequence = new ulong[sequenceLength];
41                 for (var i = 0; i < sequenceLength; i++)
42                 {
43                     sequence[i] = links.Create();
44                 }
45
46                 var sw1 = Stopwatch.StartNew();
47                 var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
48
49                 var sw2 = Stopwatch.StartNew();
50                 var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
51
52                 Assert.True(results1.Count > results2.Length);
53                 Assert.True(sw1.Elapsed > sw2.Elapsed);
54
55                 for (var i = 0; i < sequenceLength; i++)
56                 {
57                     links.Delete(sequence[i]);
58                 }
59
60                 Assert.True(links.Count() == 0);
61             }
62
63             //[Fact]
64             //public void CUDTest()
65             //{
66             //    var tempFilename = Path.GetTempFileName();
67
68             //    const long sequenceLength = 8;
69
70             //    const ulong itself = LinksConstants.Itself;
71
72             //    using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
73             //        ↳ DefaultLinksSizeStep))
74             //    {
75             //        using (var links = new Links(memoryAdapter))
76             //        {
77             //            var sequence = new ulong[sequenceLength];
78             //            for (var i = 0; i < sequenceLength; i++)
79             //            {
80             //                sequence[i] = links.Create(itself, itself);
81             //            }
82
83             //            SequencesOptions o = new SequencesOptions();
84
85             //            TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
86             //            o.
87
88             //            var sequences = new Sequences(links);
89
90             //            var sw1 = Stopwatch.StartNew();
91             //            var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
92
93             //            var sw2 = Stopwatch.StartNew();
94             //            var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
95
96             //            Assert.True(results1.Count > results2.Length);
97             //            Assert.True(sw1.Elapsed > sw2.Elapsed);
98
99             //            for (var i = 0; i < sequenceLength; i++)

```

```

96         links.Delete(sequence[i]);
97     }
98
99     File.Delete(tempFilename);
100 }
101
102 [Fact]
103 public static void AllVariantsSearchTest()
104 {
105     const long sequenceLength = 8;
106
107     using (var scope = new TempLinksTestScope(useSequences: true))
108     {
109         var links = scope.Links;
110         var sequences = scope.Sequences;
111
112         var sequence = new ulong[sequenceLength];
113         for (var i = 0; i < sequenceLength; i++)
114         {
115             sequence[i] = links.Create();
116         }
117
118         var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
119
120         //for (int i = 0; i < createResults.Length; i++)
121         //    sequences.Create(createResults[i]);
122
123         var sw0 = Stopwatch.StartNew();
124         var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
125
126         var sw1 = Stopwatch.StartNew();
127         var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
128
129         var sw2 = Stopwatch.StartNew();
130         var searchResults2 = sequences.Each1(sequence); sw2.Stop();
131
132         var sw3 = Stopwatch.StartNew();
133         var searchResults3 = sequences.Each(sequence.ShiftRight()); sw3.Stop();
134
135         var intersection0 = createResults.Intersect(searchResults0).ToList();
136         Assert.True(intersection0.Count == searchResults0.Count);
137         Assert.True(intersection0.Count == createResults.Length);
138
139         var intersection1 = createResults.Intersect(searchResults1).ToList();
140         Assert.True(intersection1.Count == searchResults1.Count);
141         Assert.True(intersection1.Count == createResults.Length);
142
143         var intersection2 = createResults.Intersect(searchResults2).ToList();
144         Assert.True(intersection2.Count == searchResults2.Count);
145         Assert.True(intersection2.Count == createResults.Length);
146
147         var intersection3 = createResults.Intersect(searchResults3).ToList();
148         Assert.True(intersection3.Count == searchResults3.Count);
149         Assert.True(intersection3.Count == createResults.Length);
150
151         for (var i = 0; i < sequenceLength; i++)
152         {
153             links.Delete(sequence[i]);
154         }
155     }
156 }
157
158 [Fact]
159 public static void BalancedVariantSearchTest()
160 {
161     const long sequenceLength = 200;
162
163     using (var scope = new TempLinksTestScope(useSequences: true))
164     {
165         var links = scope.Links;
166         var sequences = scope.Sequences;
167
168         var sequence = new ulong[sequenceLength];
169         for (var i = 0; i < sequenceLength; i++)
170         {
171             sequence[i] = links.Create();
172         }
173
174         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
175

```

```

176     var sw1 = Stopwatch.StartNew();
177     var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
178
179     var sw2 = Stopwatch.StartNew();
180     var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
181
182     var sw3 = Stopwatch.StartNew();
183     var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
184
185     // На количестве в 200 элементов это будет занимать вечность
186     //var sw4 = Stopwatch.StartNew();
187     //var searchResults4 = sequences.Each(sequence); sw4.Stop();
188
189     Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
190
191     Assert.True(searchResults3.Count == 1 && balancedVariant ==
192         ↪ searchResults3.First());
193
194     //Assert.True(sw1.Elapsed < sw2.Elapsed);
195
196     for (var i = 0; i < sequenceLength; i++)
197     {
198         links.Delete(sequence[i]);
199     }
200 }
201
202 [Fact]
203 public static void AllPartialVariantsSearchTest()
204 {
205     const long sequenceLength = 8;
206
207     using (var scope = new TempLinksTestScope(useSequences: true))
208     {
209         var links = scope.Links;
210         var sequences = scope.Sequences;
211
212         var sequence = new ulong[sequenceLength];
213         for (var i = 0; i < sequenceLength; i++)
214         {
215             sequence[i] = links.Create();
216         }
217
218         var createResults = sequences.CreateAllVariants2(sequence);
219
220         //var createResultsStrings = createResults.Select(x => x + ": " +
221             ↪ sequences.FormatSequence(x)).ToList();
222         //Global.Trash = createResultsStrings;
223
224         var partialSequence = new ulong[sequenceLength - 2];
225         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
226
227         var sw1 = Stopwatch.StartNew();
228         var searchResults1 =
229             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
230
231         var sw2 = Stopwatch.StartNew();
232         var searchResults2 =
233             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
234
235         //var sw3 = Stopwatch.StartNew();
236         //var searchResults3 =
237             ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
238
239         var sw4 = Stopwatch.StartNew();
240         var searchResults4 =
241             ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
242
243         //Global.Trash = searchResults3;
244
245         //var searchResults1Strings = searchResults1.Select(x => x + ": " +
246             ↪ sequences.FormatSequence(x)).ToList();
247         //Global.Trash = searchResults1Strings;
248
249         var intersection1 = createResults.Intersect(searchResults1).ToList();
250         Assert.True(intersection1.Count == createResults.Length);
251
252         var intersection2 = createResults.Intersect(searchResults2).ToList();

```

```

248     Assert.True(intersection2.Count == createResults.Length);
249
250     var intersection4 = createResults.Intersect(searchResults4).ToList();
251     Assert.True(intersection4.Count == createResults.Length);
252
253     for (var i = 0; i < sequenceLength; i++)
254     {
255         links.Delete(sequence[i]);
256     }
257 }
258
259 [Fact]
260 public static void BalancedPartialVariantsSearchTest()
261 {
262     const long sequenceLength = 200;
263
264     using (var scope = new TempLinksTestScope(useSequences: true))
265     {
266         var links = scope.Links;
267         var sequences = scope.Sequences;
268
269         var sequence = new ulong[sequenceLength];
270         for (var i = 0; i < sequenceLength; i++)
271         {
272             sequence[i] = links.Create();
273         }
274
275         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
276         var balancedVariant = balancedVariantConverter.Convert(sequence);
277
278         var partialSequence = new ulong[sequenceLength - 2];
279
280         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
281
282         var sw1 = Stopwatch.StartNew();
283         var searchResults1 =
284             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
285
286         var sw2 = Stopwatch.StartNew();
287         var searchResults2 =
288             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
289
290         Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
291
292         Assert.True(searchResults2.Count == 1 && balancedVariant ==
293             ↪ searchResults2.First());
294
295         for (var i = 0; i < sequenceLength; i++)
296         {
297             links.Delete(sequence[i]);
298         }
299     }
300
301     [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
332 Assert.True(matchedSequences1.Count == 0);
333
334 var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
335
336 Assert.True(matchedSequences2.Count == 0);
337
338 var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
339
340 Assert.True(matchedSequences3.Count == 0);
341
342 var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
343
344 Assert.Contains(doublet, matchedSequences4);
345 Assert.Contains(balancedVariant, matchedSequences4);
346
347 for (var i = 0; i < sequence.Length; i++)
348 {
349     links.Delete(sequence[i]);
350 }
351 }
352 }
353
354 [Fact]
355 public static void IndexTest()
356 {
357     using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
358         ↪ true }, useSequences: true))
359     {
360         var links = scope.Links;
361         var sequences = scope.Sequences;
362         var index = sequences.Options.Index;
363
364         var e1 = links.Create();
365         var e2 = links.Create();
366
367         var sequence = new[]
368         {
369             e1, e2, e1, e2 // mama / papa
370         };
371
372         Assert.False(index.MightContain(sequence));
373
374         index.Add(sequence);
375
376         Assert.True(index.MightContain(sequence));
377     }
378
379     /// <summary>Imported from https://raw.githubusercontent.com/Konard/LinksPlatform/%
380     ↪ %D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87
381     ↪ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
382     private static readonly string _exampleText =
383     ↪ @"([english
384     ↪ version] (https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))

```

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

```

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

```

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

```

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

```

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

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

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

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

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

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

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

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

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

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

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

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

414  
415 Допустим у нас есть смысл "связать" и смысл "направления", много ли это нам даёт? Много ли  
→ вариантов интерпретаций? 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,
↳ totalSequenceSymbolFrequencyCounter);
458         var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
↳ balancedVariantConverter, doubletFrequenciesCache);
459
460         var compressedVariant = compressingConverter.Convert(sequence);
461
462         // 1: [1] (1->1) point
463         // 2: [2] (2->2) point
464         // 3: [1,2] (1->2) doublet
465         // 4: [1,2,1,2] (3->3) doublet
466
467         Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
468         Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
469         Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
470         Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
471
472         var source = _constants.SourcePart;
473         var target = _constants.TargetPart;
474
475         Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
476         Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
477         Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
478         Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
479
480         // 4 - length of sequence

```



```

481     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
482         ↪ == sequence[0]);
483     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
484         ↪ == sequence[1]);
485     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
486         ↪ == sequence[2]);
487     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
488         ↪ == sequence[3]);
489 }
490
491 [Fact]
492 public static void CompressionEfficiencyTest()
493 {
494     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
495         ↪ StringSplitOptions.RemoveEmptyEntries);
496     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
497     var totalCharacters = arrays.Select(x => x.Length).Sum();
498
499     using (var scope1 = new TempLinksTestScope(useSequences: true))
500     using (var scope2 = new TempLinksTestScope(useSequences: true))
501     using (var scope3 = new TempLinksTestScope(useSequences: true))
502     {
503         scope1.Links.Unsync.UseUnicode();
504         scope2.Links.Unsync.UseUnicode();
505         scope3.Links.Unsync.UseUnicode();
506
507         var balancedVariantConverter1 = new
508             ↪ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
509         var totalSequenceSymbolFrequencyCounter = new
510             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
511         var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
512             ↪ totalSequenceSymbolFrequencyCounter);
513         var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
514             ↪ balancedVariantConverter1, linkFrequenciesCache1,
515             ↪ doInitialFrequenciesIncrement: false);
516
517         //var compressor2 = scope2.Sequences;
518         var compressor3 = scope3.Sequences;
519
520         var constants = Default<LinksConstants<ulong>>.Instance;
521
522         var sequences = compressor3;
523         //var meaningRoot = links.CreatePoint();
524         //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
525         //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
526         //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
527             ↪ constants.Itself);
528
529         //var unaryNumberToAddressConverter = new
530             ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
531         //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
532             ↪ unaryOne);
533         //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
534             ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
535         //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
536             ↪ frequencyPropertyMarker, frequencyMarker);
537         //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
538             ↪ frequencyPropertyOperator, frequencyIncrementer);
539         //var linkToItsFrequencyNumberConverter = new
540             ↪ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
541             ↪ unaryNumberToAddressConverter);
542
543         var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
544             ↪ totalSequenceSymbolFrequencyCounter);
545
546         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache3);
547
548         var sequenceToItsLocalElementLevelsConverter = new
549             ↪ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
550             ↪ linkToItsFrequencyNumberConverter);
551         var optimalVariantConverter = new
552             ↪ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
553             ↪ sequenceToItsLocalElementLevelsConverter);
554
555         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
556     ↳ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
557
558 var initialCount2 = scope2.Links.Unsync.Count();
559
560 var sw2 = Stopwatch.StartNew();
561
562 for (int i = START; i < END; i++)
563 {
564     compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
565 }
566
567 var elapsed2 = sw2.Elapsed;
568
569 for (int i = START; i < END; i++)
570 {
571     linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
572 }
573
574 var initialCount3 = scope3.Links.Unsync.Count();
575
576 var sw3 = Stopwatch.StartNew();
577
578 for (int i = START; i < END; i++)
579 {
580     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
581     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
582 }
583
584 var elapsed3 = sw3.Elapsed;
585
586 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
587     ↳ Optimal variant: {elapsed3}");
588
589 // Assert.True(elapsed1 > elapsed2);
590
591 // Checks
592 for (int i = START; i < END; i++)
593 {
594     var sequence1 = compressed1[i];
595     var sequence2 = compressed2[i];
596     var sequence3 = compressed3[i];
597
598     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
599         ↳ scope1.Links.Unsync);
600
601     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
602         ↳ scope2.Links.Unsync);
603
604     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
605         ↳ scope3.Links.Unsync);
606
607     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
608         ↳ link.IsPartialPoint());
609     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
610         ↳ link.IsPartialPoint());
611     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
612         ↳ link.IsPartialPoint());

```

```

605         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
606         ↪ arrays[i].Length > 3)
607         //    Assert.False(structure1 == structure2);
608         //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
609         ↪ arrays[i].Length > 3)
610         //    Assert.False(structure3 == structure2);
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) <
616     ↪ totalCharacters);
617     Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
618     ↪ totalCharacters);
619     Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
620     ↪ totalCharacters);
621
622     Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
623     ↪ totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
624     ↪ totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
625     ↪ totalCharacters}");
626
627     Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
628     ↪ scope2.Links.Unsync.Count() - initialCount2);
629     Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
630     ↪ scope2.Links.Unsync.Count() - initialCount2);
631
632     var duplicateProvider1 = new
633     ↪ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
634     var duplicateProvider2 = new
635     ↪ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
636     var duplicateProvider3 = new
637     ↪ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
638
639     var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
640     var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
641     var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
642
643     var duplicates1 = duplicateCounter1.Count();
644
645     ConsoleHelpers.Debug("-----");
646
647     var duplicates2 = duplicateCounter2.Count();
648
649     ConsoleHelpers.Debug("-----");
650
651     var duplicates3 = duplicateCounter3.Count();
652
653     Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
654
655     linkFrequenciesCache1.ValidateFrequencies();
656     linkFrequenciesCache3.ValidateFrequencies();
657 }
658
659 [Fact]
660 public static void CompressionStabilityTest()
661 {
662     // TODO: Fix bug (do a separate test)
663     //const ulong minNumbers = 0;
664     //const ulong maxNumbers = 1000;
665
666     const ulong minNumbers = 10000;
667     const ulong maxNumbers = 12500;
668
669     var strings = new List<string>();
670
671     for (ulong i = minNumbers; i < maxNumbers; i++)
672     {
673         strings.Add(i.ToString());
674     }
675
676     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
677     var totalCharacters = arrays.Select(x => x.Length).Sum();
678

```

```

669 using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
    ↳ SequencesOptions<ulong> { UseCompression = true,
    ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
670 using (var scope2 = new TempLinksTestScope(useSequences: true))
671 {
672     scope1.Links.UseUnicode();
673     scope2.Links.UseUnicode();
674
675     //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
676     var compressor1 = scope1.Sequences;
677     var compressor2 = scope2.Sequences;
678
679     var compressed1 = new ulong[arrays.Length];
680     var compressed2 = new ulong[arrays.Length];
681
682     var sw1 = Stopwatch.StartNew();
683
684     var START = 0;
685     var END = arrays.Length;
686
687     // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
688     // Stability issue starts at 10001 or 11000
689     //for (int i = START; i < END; i++)
690     //{
691     //    var first = compressor1.Compress(arrays[i]);
692     //    var second = compressor1.Compress(arrays[i]);
693
694     //    if (first == second)
695     //        compressed1[i] = first;
696     //    else
697     //    {
698     //        // TODO: Find a solution for this case
699     //    }
700     //}
701
702     for (int i = START; i < END; i++)
703     {
704         var first = compressor1.Create(arrays[i].ShiftRight());
705         var second = compressor1.Create(arrays[i].ShiftRight());
706
707         if (first == second)
708         {
709             compressed1[i] = first;
710         }
711         else
712         {
713             // TODO: Find a solution for this case
714         }
715     }
716
717     var elapsed1 = sw1.Elapsed;
718
719     var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
720
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:
    ↳ {elapsed2}");
737
738     Assert.True(elapsed1 > elapsed2);
739
740     // Checks
741     for (int i = START; i < END; i++)
742     {
743         var sequence1 = compressed1[i];
744         var sequence2 = compressed2[i];

```

```

745         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
746         {
747             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
748                 ↪ scope1.Links);
749
750             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
751                 ↪ scope2.Links);
752
753             //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
754             ↪ link.IsPartialPoint());
755             //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
756             ↪ link.IsPartialPoint());
757
758             //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
759             ↪ arrays[i].Length > 3)
760             //    Assert.False(structure1 == structure2);
761
762             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
763         }
764     }
765
766     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
767     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
768
769     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
770     ↪ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
771     ↪ totalCharacters}}");
772
773     Assert.True(scope1.Links.Count() <= scope2.Links.Count());
774
775     //compressor1.ValidateFrequencies();
776 }
777
778 [Fact]
779 public static void RandomNumbersCompressionQualityTest()
780 {
781     const ulong N = 500;
782
783     //const ulong minNumbers = 10000;
784     //const ulong maxNumbers = 20000;
785
786     //var strings = new List<string>();
787
788     //for (ulong i = 0; i < N; i++)
789     //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
790     ↪ maxNumbers).ToString());
791
792     var strings = new List<string>();
793
794     for (ulong i = 0; i < N; i++)
795     {
796         strings.Add(RandomHelpers.Default.NextUInt64().ToString());
797     }
798
799     strings = strings.Distinct().ToList();
800
801     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
802     var totalCharacters = arrays.Select(x => x.Length).Sum();
803
804     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
805     ↪ SequencesOptions<ulong> { UseCompression = true,
806     ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
807     using (var scope2 = new TempLinksTestScope(useSequences: true))
808     {
809         scope1.Links.UseUnicode();
810         scope2.Links.UseUnicode();
811
812         var compressor1 = scope1.Sequences;
813         var compressor2 = scope2.Sequences;
814
815         var compressed1 = new ulong[arrays.Length];
816         var compressed2 = new ulong[arrays.Length];
817
818         var sw1 = Stopwatch.StartNew();
819
820         var START = 0;
821         var END = arrays.Length;

```

```

814     for (int i = START; i < END; i++)
815     {
816         compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
817     }
818
819     var elapsed1 = sw1.Elapsed;
820
821     var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
822
823     var sw2 = Stopwatch.StartNew();
824
825     for (int i = START; i < END; i++)
826     {
827         compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
828     }
829
830     var elapsed2 = sw2.Elapsed;
831
832     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
833     ↪ {elapsed2}");
834
835     Assert.True(elapsed1 > elapsed2);
836
837     // Checks
838     for (int i = START; i < END; i++)
839     {
840         var sequence1 = compressed1[i];
841         var sequence2 = compressed2[i];
842
843         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
844         {
845             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
846             ↪ scope1.Links);
847
848             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
849             ↪ scope2.Links);
850
851             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
852         }
853     }
854
855     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
856     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
857
858     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
859     ↪ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
860     ↪ totalCharacters}}");
861
862     // Can be worse than balanced variant
863     //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
864
865     //compressor1.ValidateFrequencies();
866 }
867
868 [Fact]
869 public static void AllTreeBreakDownAtSequencesCreationBugTest()
870 {
871     // Made out of AllPossibleConnectionsTest test.
872
873     //const long sequenceLength = 5; //100% bug
874     const long sequenceLength = 4; //100% bug
875     //const long sequenceLength = 3; //100% _no_bug (ok)
876
877     using (var scope = new TempLinksTestScope(useSequences: true))
878     {
879         var links = scope.Links;
880         var sequences = scope.Sequences;
881
882         var sequence = new ulong[sequenceLength];
883         for (var i = 0; i < sequenceLength; i++)
884         {
885             sequence[i] = links.Create();
886         }
887
888         var createResults = sequences.CreateAllVariants2(sequence);
889
890         Global.Trash = createResults;
891     }
892 }

```

```

888         for (var i = 0; i < sequenceLength; i++)
889         {
890             links.Delete(sequence[i]);
891         }
892     }
893 }
894
895 [Fact]
896 public static void AllPossibleConnectionsTest()
897 {
898     const long sequenceLength = 5;
899
900     using (var scope = new TempLinksTestScope(useSequences: true))
901     {
902         var links = scope.Links;
903         var sequences = scope.Sequences;
904
905         var sequence = new ulong[sequenceLength];
906         for (var i = 0; i < sequenceLength; i++)
907         {
908             sequence[i] = links.Create();
909         }
910
911         var createResults = sequences.CreateAllVariants2(sequence);
912         var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
913
914         for (var i = 0; i < 1; i++)
915         {
916             var sw1 = Stopwatch.StartNew();
917             var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
918
919             var sw2 = Stopwatch.StartNew();
920             var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
921
922             var sw3 = Stopwatch.StartNew();
923             var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
924
925             var sw4 = Stopwatch.StartNew();
926             var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
927
928             Global.Trash = searchResults3;
929             Global.Trash = searchResults4; //-V3008
930
931             var intersection1 = createResults.Intersect(searchResults1).ToList();
932             Assert.True(intersection1.Count == createResults.Length);
933
934             var intersection2 = reverseResults.Intersect(searchResults1).ToList();
935             Assert.True(intersection2.Count == reverseResults.Length);
936
937             var intersection0 = searchResults1.Intersect(searchResults2).ToList();
938             Assert.True(intersection0.Count == searchResults2.Count);
939
940             var intersection3 = searchResults2.Intersect(searchResults3).ToList();
941             Assert.True(intersection3.Count == searchResults3.Count);
942
943             var intersection4 = searchResults3.Intersect(searchResults4).ToList();
944             Assert.True(intersection4.Count == searchResults4.Count);
945         }
946
947         for (var i = 0; i < sequenceLength; i++)
948         {
949             links.Delete(sequence[i]);
950         }
951     }
952 }
953
954 [Fact(Skip = "Correct implementation is pending")]
955 public static void CalculateAllUsagesTest()
956 {
957     const long sequenceLength = 3;
958
959     using (var scope = new TempLinksTestScope(useSequences: true))
960     {
961         var links = scope.Links;
962         var sequences = scope.Sequences;
963
964         var sequence = new ulong[sequenceLength];
965         for (var i = 0; i < sequenceLength; i++)
966         {
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 }

```

#### 1.179 ./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  using Platform.Data.Doublets.Memory;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public unsafe static class SplitMemoryGenericLinksTests
10     {
11         [Fact]
12         public static void CRUDTest()
13         {
14             Using<byte>(links => links.TestCRUDOperations());
15             Using<ushort>(links => links.TestCRUDOperations());
16             Using<uint>(links => links.TestCRUDOperations());
17             Using<ulong>(links => links.TestCRUDOperations());
18         }
19
20         [Fact]
21         public static void RawNumbersCRUDTest()
22         {
23             UsingWithExternalReferences<byte>(links => links.TestRawNumbersCRUDOperations());
24             UsingWithExternalReferences<ushort>(links => links.TestRawNumbersCRUDOperations());
25             UsingWithExternalReferences<uint>(links => links.TestRawNumbersCRUDOperations());
26             UsingWithExternalReferences<ulong>(links => links.TestRawNumbersCRUDOperations());
27         }
28
29         [Fact]
30         public static void MultipleRandomCreationsAndDeletionsTest()
31         {
32             Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
33             ↪ MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
34             ↪ implementation of tree cuts out 5 bits from the address space.
35             Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
36             ↪ stMultipleRandomCreationsAndDeletions(100));
37             Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
38             ↪ MultipleRandomCreationsAndDeletions(100));
39             Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
40             ↪ tMultipleRandomCreationsAndDeletions(100));
41         }
42
43         private static void Using<TLink>(Action<ILinks<TLink>> action)
44         {
45             using (var dataMemory = new HeapResizableDirectMemory())
46             using (var indexMemory = new HeapResizableDirectMemory())
47             using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory))

```



```

43     {
44         action(memory);
45     }
46 }
47
48 private static void UsingWithExternalReferences<TLink>(Action<ILinks<TLink>> action)
49 {
50     var constants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
51     using (var dataMemory = new HeapResizableDirectMemory())
52     using (var indexMemory = new HeapResizableDirectMemory())
53     using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory,
54         ↪ SplitMemoryLinks<TLink>.DefaultLinksSizeStep, constants))
55     {
56         action(memory);
57     }
58 }
59 }

```

#### 1.180 ./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().TestMultipleRandomCreationsAndDeletions(500));
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     }

```

#### 1.181 ./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(500));
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    }

```

### 1.182 ./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 using Platform.Data.Doublets.Memory.Split.Specific;
7 using Platform.Memory;
8
9 namespace Platform.Data.Doublets.Tests
10 {
11     public class TempLinksTestScope : DisposableBase
12     {
13         public ILinks<ulong> MemoryAdapter { get; }
14         public SynchronizedLinks<ulong> Links { get; }
15         public Sequences.Sequences Sequences { get; }
16         public string TempFilename { get; }
17         public string TempTransactionLogFilename { get; }
18         private readonly bool _deleteFiles;
19
20         public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
21             ↪ useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
22             ↪ useLog) { }
23
24         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
25             ↪ true, bool useSequences = false, bool useLog = false)
26         {
27             _deleteFiles = deleteFiles;
28             TempFilename = Path.GetTempFileName();
29             TempTransactionLogFilename = Path.GetTempFileName();

```

```

27 //var coreMemoryAdapter = new UInt64UnitedMemoryLinks(TempFilename);
28 var coreMemoryAdapter = new UInt64SplitMemoryLinks(new
    ↳ FileMappedResizableDirectMemory(TempFilename), new
    ↳ FileMappedResizableDirectMemory(Path.ChangeExtension(TempFilename, "indexes")),
    ↳ UInt64SplitMemoryLinks.DefaultLinksSizeStep, new LinksConstants<ulong>(),
    ↳ Memory.IndexTreeType.Default, useLinkedList: true);
29 MemoryAdapter = useLog ? (ILinks<ulong>)new
    ↳ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
    ↳ coreMemoryAdapter;
30 Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
31 if (useSequences)
32 {
33     Sequences = new Sequences.Sequences(Links, sequencesOptions);
34 }
35 }
36
37 protected override void Dispose(bool manual, bool wasDisposed)
38 {
39     if (!wasDisposed)
40     {
41         Links.Unsync.DisposeIfPossible();
42         if (_deleteFiles)
43         {
44             DeleteFiles();
45         }
46     }
47 }
48
49 public void DeleteFiles()
50 {
51     File.Delete(TempFilename);
52     File.Delete(TempTransactionLogFilename);
53 }
54 }
55 }

```

### 1.183 ./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs

```

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

```

```

43     links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
44
45     Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
46
47     // Update link to reference itself
48     links.Update(linkAddress, linkAddress, linkAddress);
49
50     link = new Link<T>(links.GetLink(linkAddress));
51
52     Assert.True(equalityComparer.Equals(link.Source, linkAddress));
53     Assert.True(equalityComparer.Equals(link.Target, linkAddress));
54
55     // Update link to reference null (prepare for delete)
56     var updated = links.Update(linkAddress, constants.Null, constants.Null);
57
58     Assert.True(equalityComparer.Equals(updated, linkAddress));
59
60     link = new Link<T>(links.GetLink(linkAddress));
61
62     Assert.True(equalityComparer.Equals(link.Source, constants.Null));
63     Assert.True(equalityComparer.Equals(link.Target, constants.Null));
64
65     // Delete link
66     links.Delete(linkAddress);
67
68     Assert.True(equalityComparer.Equals(links.Count(), zero));
69
70     setter = new Setter<T>(constants.Null);
71     links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
72
73     Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
74 }
75
76 public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
77 {
78     // Constants
79     var constants = links.Constants;
80     var equalityComparer = EqualityComparer<T>.Default;
81
82     var zero = default(T);
83     var one = Arithmetic.Increment(zero);
84     var two = Arithmetic.Increment(one);
85
86     var h106E = new Hybrid<T>(106L, isExternal: true);
87     var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
88     var h108E = new Hybrid<T>(-108L);
89
90     Assert.Equal(106L, h106E.AbsoluteValue);
91     Assert.Equal(107L, h107E.AbsoluteValue);
92     Assert.Equal(108L, h108E.AbsoluteValue);
93
94     // Create Link (External -> External)
95     var linkAddress1 = links.Create();
96
97     links.Update(linkAddress1, h106E, h108E);
98
99     var link1 = new Link<T>(links.GetLink(linkAddress1));
100
101     Assert.True(equalityComparer.Equals(link1.Source, h106E));
102     Assert.True(equalityComparer.Equals(link1.Target, h108E));
103
104     // Create Link (Internal -> External)
105     var linkAddress2 = links.Create();
106
107     links.Update(linkAddress2, linkAddress1, h108E);
108
109     var link2 = new Link<T>(links.GetLink(linkAddress2));
110
111     Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
112     Assert.True(equalityComparer.Equals(link2.Target, h108E));
113
114     // Create Link (Internal -> Internal)
115     var linkAddress3 = links.Create();
116
117     links.Update(linkAddress3, linkAddress1, linkAddress2);
118
119     var link3 = new Link<T>(links.GetLink(linkAddress3));
120
121     Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
122     Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));

```

```

123
124 // Search for created link
125 var setter1 = new Setter<T>(constants.Null);
126 links.Each(h106E, h108E, setter1.SetAndReturnFalse);
127
128 Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
129
130 // Search for nonexistent link
131 var setter2 = new Setter<T>(constants.Null);
132 links.Each(h106E, h107E, setter2.SetAndReturnFalse);
133
134 Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
135
136 // Update link to reference null (prepare for delete)
137 var updated = links.Update(linkAddress3, constants.Null, constants.Null);
138
139 Assert.True(equalityComparer.Equals(updated, linkAddress3));
140
141 link3 = new Link<T>(links.GetLink(linkAddress3));
142
143 Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
144 Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
145
146 // Delete link
147 links.Delete(linkAddress3);
148
149 Assert.True(equalityComparer.Equals(links.Count(), two));
150
151 var setter3 = new Setter<T>(constants.Null);
152 links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
153
154 Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
155 }
156
157 public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
→ links, int maximumOperationsPerCycle)
158 {
159     var comparer = Comparer<TLink>.Default;
160     var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
161     var uint64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
162     for (var N = 1; N < maximumOperationsPerCycle; N++)
163     {
164         var random = new System.Random(N);
165         var created = 0UL;
166         var deleted = 0UL;
167         for (var i = 0; i < N; i++)
168         {
169             var linksCount = addressToUInt64Converter.Convert(links.Count());
170             var createPoint = random.NextBoolean();
171             if (linksCount >= 2 && createPoint)
172             {
173                 var linksAddressRange = new Range<ulong>(1, linksCount);
174                 TLink source = uint64ToAddressConverter.Convert(random.NextUInt64(linksA_
→ ddressRange));
175                 TLink target = uint64ToAddressConverter.Convert(random.NextUInt64(linksA_
→ ddressRange));
176                 → //-V3086
177                 var resultLink = links.GetOrCreate(source, target);
178                 if (comparer.Compare(resultLink,
179                 → uint64ToAddressConverter.Convert(linksCount)) > 0)
180                 {
181                     created++;
182                 }
183             }
184             else
185             {
186                 links.Create();
187                 created++;
188             }
189         }
190         Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
191         for (var i = 0; i < N; i++)
192         {
193             TLink link = uint64ToAddressConverter.Convert((ulong)i + 1UL);
194             if (links.Exists(link))
195             {
196                 links.Delete(link);
197                 deleted++;
198             }
199         }
200     }

```



```

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
            ↪ tion>(scope.TempTransactionLogFilename);
138
139                 l2 = links.Update(l2, l1);
140
141                 links.Delete(l2);
142
143                 ExceptionThrower();

```

```

144         transaction.Commit();
145     }
146
147     Global.Trash = links.Count();
148 }
149
150 }
151 catch
152 {
153     Assert.False(lastScope == null);
154
155     var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
        ↪ astScope.TempTransactionLogFilename);
156
157     Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
        ↪ transitions[0].After.IsNull());
158
159     lastScope.DeleteFiles();
160 }
161
162 }
163
164 [Fact]
165 public static void TransactionUserCodeErrorSomeDataSavedTest()
166 {
167     // User Code Error (Autoreverted), some data saved
168     var itself = _constants.Itself;
169
170     TempLinksTestScope lastScope = null;
171     try
172     {
173         ulong l1;
174         ulong l2;
175
176         using (var scope = new TempLinksTestScope(useLog: true))
177         {
178             var links = scope.Links;
179             l1 = links.CreateAndUpdate(itself, itself);
180             l2 = links.CreateAndUpdate(itself, itself);
181
182             l2 = links.Update(l2, l2, l1, l2);
183
184             links.CreateAndUpdate(l2, itself);
185             links.CreateAndUpdate(l2, itself);
186
187             links.Unsync.DisposeIfPossible();
188
189             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
                ↪ scope.TempTransactionLogFilename);
190
191         }
192
193         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
            ↪ useLog: true))
194         {
195             var links = scope.Links;
196             var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
197             using (var transaction = transactionsLayer.BeginTransaction())
198             {
199                 l2 = links.Update(l2, l1);
200
201                 links.Delete(l2);
202
203                 ExceptionThrower();
204
205                 transaction.Commit();
206             }
207
208             Global.Trash = links.Count();
209         }
210     }
211     catch
212     {
213         Assert.False(lastScope == null);
214
215         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
            ↪ Scope.TempTransactionLogFilename);
216
217         lastScope.DeleteFiles();
218     }
219 }

```



```

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) ==
403             ↪ "(5:(4:5 (6:5 4)) 6)");
404         Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
405             ↪ "(6:(5:(4:5 6) 6) 4)");
406         Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
407             ↪ "(4:(5:4 (6:5 4)) 6)");
408
409         // TODO: Think how to build balanced syntax tree while formatting structure (eg.
410         ↪ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)"
411
412         Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
413             ↪ "{5}{5}{4}{6}");
414         Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
415             ↪ "{5}{6}{6}{4}");
416         Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
417             ↪ "{4}{5}{4}{6}");
418     }
419 }
420
421 private static void DefaultFormatter(StringBuilder sb, ulong link)
422 {
423     sb.Append(link.ToString());
424 }
425
426 #endregion
427
428 #region Performance
429
430 /*
431 public static void RunAllPerformanceTests()
432 {
433     try
434     {
435         links.TestLinksInSteps();
436     }
437     catch (Exception ex)
438     {
439         ex.WriteToConsole();
440     }
441
442     return;
443 }
444 */

```

```

437     try
438     {
439         //ThreadPool.SetMaxThreads(2, 2);
440
441         // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
↪ результат
442         // Также это дополнительно помогает в отладке
443         // Увеличивает вероятность попадания информации в кэши
444         for (var i = 0; i < 10; i++)
445         {
446             //0 - 10 ГБ
447             //Каждые 100 МБ срез цифр
448
449             //links.TestGetSourceFunction();
450             //links.TestGetSourceFunctionInParallel();
451             //links.TestGetTargetFunction();
452             //links.TestGetTargetFunctionInParallel();
453             links.Create64BillionLinks();
454
455             links.TestRandomSearchFixed();
456             //links.Create64BillionLinksInParallel();
457             links.TestEachFunction();
458             //links.TestForeach();
459             //links.TestParallelForeach();
460         }
461
462         links.TestDeletionOfAllLinks();
463
464     }
465     catch (Exception ex)
466     {
467         ex.WriteToConsole();
468     }
469 }*/
470
471 /*
472 public static void TestLinksInSteps()
473 {
474     const long gibibyte = 1024 * 1024 * 1024;
475     const long mebibyte = 1024 * 1024;
476
477     var totalLinksToCreate = gibibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
478     var linksStep = 102 * mebibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
479
480     var creationMeasurements = new List<TimeSpan>();
481     var searchMeasurements = new List<TimeSpan>();
482     var deletionMeasurements = new List<TimeSpan>();
483
484     GetBaseRandomLoopOverhead(linksStep);
485     GetBaseRandomLoopOverhead(linksStep);
486
487     var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
488
489     ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
490
491     var loops = totalLinksToCreate / linksStep;
492
493     for (int i = 0; i < loops; i++)
494     {
495         creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
496         searchMeasurements.Add(Measure(() => links.RunRandomSearches(linksStep)));
497
498         Console.WriteLine("\rC + S {0}/{1}", i + 1, loops);
499     }
500
501     ConsoleHelpers.Debug();
502
503     for (int i = 0; i < loops; i++)
504     {
505         deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
506
507         Console.WriteLine("\rD {0}/{1}", i + 1, loops);
508     }
509
510     ConsoleHelpers.Debug();
511
512     ConsoleHelpers.Debug("C S D");
513

```

```

514         for (int i = 0; i < loops; i++)
515         {
516             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
↵ searchMeasurements[i], deletionMeasurements[i]);
517         }
518
519         ConsoleHelpers.Debug("C S D (no overhead)");
520
521         for (int i = 0; i < loops; i++)
522         {
523             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
↵ searchMeasurements[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
524         }
525
526         ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
↵ links.Total);
527     }
528
529     private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
↵ amountToCreate)
530     {
531         for (long i = 0; i < amountToCreate; i++)
532             links.Create(0, 0);
533     }
534
535     private static TimeSpan GetBaseRandomLoopOverhead(long loops)
536     {
537         return Measure(() =>
538         {
539             ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
540             ulong result = 0;
541             for (long i = 0; i < loops; i++)
542             {
543                 var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
544                 var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
545
546                 result += maxValue + source + target;
547             }
548             Global.Trash = result;
549         });
550     }
551     */
552
553     [Fact(Skip = "performance test")]
554     public static void GetSourceTest()
555     {
556         using (var scope = new TempLinksTestScope())
557         {
558             var links = scope.Links;
559             ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
↵ 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}",
584                 Iterations, elapsedTime, (long)iterationsPerSecond, counter);
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
594             ↳ parallel.", Iterations);
595
596         long counter = 0;
597
598         //var firstLink = links.First();
599         var firstLink = links.Create();
600
601         var sw = Stopwatch.StartNew();
602
603         // Тестируем саму функцию
604         Parallel.For(0, Iterations, x =>
605         {
606             Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
607             //Interlocked.Increment(ref counter);
608         });
609
610         var elapsedTime = sw.Elapsed;
611
612         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
613
614         links.Delete(firstLink);
615
616         ConsoleHelpers.Debug(
617             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
618             ↳ second), counter result: {3}",
619             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
620     }
621 }
622 [Fact(Skip = "performance test")]
623 public static void TestGetTarget()
624 {
625     using (var scope = new TempLinksTestScope())
626     {
627         var links = scope.Links;
628         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
629             ↳ Iterations);
630
631         ulong counter = 0;
632
633         //var firstLink = links.First();
634         var firstLink = links.Create();
635
636         var sw = Stopwatch.StartNew();
637
638         for (ulong i = 0; i < Iterations; i++)
639         {
640             counter += links.GetTarget(firstLink);
641         }
642
643         var elapsedTime = sw.Elapsed;
644
645         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
646
647         links.Delete(firstLink);
648
649         ConsoleHelpers.Debug(
650             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
651             ↳ second), counter result: {3}",
652             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
653     }
654 }
655 [Fact(Skip = "performance test")]
656 public static void TestGetTargetInParallel()
657 {
658     using (var scope = new TempLinksTestScope())
659     {
660         var links = scope.Links;
661         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
662             ↳ parallel.", Iterations);

```

```

661         long counter = 0;
662
663
664         //var firstLink = links.First();
665         var firstLink = links.Create();
666
667         var sw = Stopwatch.StartNew();
668
669         Parallel.For(0, Iterations, x =>
670         {
671             Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
672             //Interlocked.Increment(ref counter);
673         });
674
675         var elapsedTime = sw.Elapsed;
676
677         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
678
679         links.Delete(firstLink);
680
681         ConsoleHelpers.Debug(
682             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        ↳ Iterations, elapsedTime, (long)iterationsPerSecond, counter);
683     }
684 }
685
686
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}
69 ↳ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
69 ↳ counter);
722     }
723
724     File.Delete(tempFilename);
725 }*/
726
727 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
728 public static void TestRandomSearchAll()
729 {
730     using (var scope = new TempLinksTestScope())
731     {
732         var links = scope.Links;
733         ulong counter = 0;
734
735         var maxLink = links.Count();
736
737         var iterations = links.Count();

```

```

734 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
735     ↪ links.Count());
736
737 var sw = Stopwatch.StartNew();
738
739 for (var i = iterations; i > 0; i--)
740 {
741     var linksAddressRange = new
742         ↪ Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);
743
744     var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
745     var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
746
747     counter += links.SearchOrDefault(source, target);
748 }
749
750 var elapsedTime = sw.Elapsed;
751
752 var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
753
754 ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
755     ↪ Iterations per second), c: {3}",
756     iterations, elapsedTime, (long)iterationsPerSecond, counter);
757 }
758
759 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
760 public static void TestEach()
761 {
762     using (var scope = new TempLinksTestScope())
763     {
764         var links = scope.Links;
765
766         var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
767
768         ConsoleHelpers.Debug("Testing Each function.");
769
770         var sw = Stopwatch.StartNew();
771
772         links.Each(counter.IncrementAndReturnTrue);
773
774         var elapsedTime = sw.Elapsed;
775
776         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
777
778         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
779             ↪ links per second)",
780             counter, elapsedTime, (long)linksPerSecond);
781     }
782 }
783
784 /*
785 [Fact]
786 public static void TestForeach()
787 {
788     var tempFilename = Path.GetTempFileName();
789
790     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
791         ↪ DefaultLinksSizeStep))
792     {
793         ulong counter = 0;
794
795         ConsoleHelpers.Debug("Testing foreach through links.");
796
797         var sw = Stopwatch.StartNew();
798
799         //foreach (var link in links)
800         //{
801             counter++;
802         //}
803
804         var elapsedTime = sw.Elapsed;
805
806         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
807
808         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
809             ↪ links per second)", counter, elapsedTime, (long)linksPerSecond);
810     }
811 }

```



```

808         File.Delete(tempFilename);
809     }
810     */
811
812     /*
813     [Fact]
814     public static void TestParallelForeach()
815     {
816         var tempFilename = Path.GetTempFileName();
817
818         using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪ DefaultLinksSizeStep))
819         {
820             long counter = 0;
821
822             ConsoleHelpers.Debug("Testing parallel foreach through links.");
823
824             var sw = Stopwatch.StartNew();
825
826             //Parallel.ForEach((IEnumerable<ulong>)links, x =>
827             //{
828             //    Interlocked.Increment(ref counter);
829             //});
830
831             var elapsedTime = sw.Elapsed;
832
833             var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
834
835             ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↪ {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
836         }
837
838         File.Delete(tempFilename);
839     }
840     */
841
842     [Fact(Skip = "performance test")]
843     public static void Create64BillionLinks()
844     {
845         using (var scope = new TempLinksTestScope())
846         {
847             var links = scope.Links;
848             var linksBeforeTest = links.Count();
849
850             long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;
851
852             ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
853
854             var elapsedTime = Performance.Measure(() =>
855             {
856                 for (long i = 0; i < linksToCreate; i++)
857                 {
858                     links.Create();
859                 }
860             });
861
862             var linksCreated = links.Count() - linksBeforeTest;
863             var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
864
865             ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
866
867             ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
↪ linksCreated, elapsedTime,
868                 (long)linksPerSecond);
869         }
870     }
871
872     [Fact(Skip = "performance test")]
873     public static void Create64BillionLinksInParallel()
874     {
875         using (var scope = new TempLinksTestScope())
876         {
877             var links = scope.Links;
878             var linksBeforeTest = links.Count();
879
880             var sw = Stopwatch.StartNew();
881
882             long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;
883
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 }

```

#### 1.185 ./csharp/Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs

```

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

```

```

33         Assert.Equal(numbers[i],
34             ↪ fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
35     }
36 }
37 }
38 }

```

## 1.186 ./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         private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
57             ↪ meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
58             ↪ numberToAddressConverter)
59         {
60             var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
61             var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
62                 ↪ addressToNumberConverter, unicodeSymbolMarker);
63             var originalCharacter = 'H';
64             var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
65             var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
66                 ↪ unicodeSymbolMarker);
67             var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
68                 ↪ numberToAddressConverter, unicodeSymbolCriterionMatcher);
69             var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
70             Assert.Equal(originalCharacter, resultingCharacter);
71         }
72     }
73 }

```

```
}
```

```
[Fact]
```

```
public static void StringAndUnicodeSequenceConvertersTest()
```

```
{
```

```
    using (var scope = new TempLinksTestScope())
```

```
    {
```

```
        var links = scope.Links;
```

```
        var itself = links.Constants.Itself;
```

```
        var meaningRoot = links.CreatePoint();
```

```
        var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
```

```
        var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
```

```
        var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
```

```
        var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
```

```
        var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
```

```
        var powerOf2ToUnaryNumberConverter = new
```

```
        ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
```

```
        var addressToUnaryNumberConverter = new
```

```
        ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
```

```
        var charToUnicodeSymbolConverter = new
```

```
        ↪ CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
```

```
        ↪ unicodeSymbolMarker);
```

```
        var unaryNumberToAddressConverter = new
```

```
        ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
```

```
        ↪ powerOf2ToUnaryNumberConverter);
```

```
        var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
```

```
        var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
```

```
        ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
```

```
        var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
```

```
        ↪ frequencyPropertyMarker, frequencyMarker);
```

```
        var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
```

```
        ↪ frequencyPropertyOperator, frequencyIncrementer);
```

```
        var linkToItsFrequencyNumberConverter = new
```

```
        ↪ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
```

```
        ↪ unaryNumberToAddressConverter);
```

```
        var sequenceToItsLocalElementLevelsConverter = new
```

```
        ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
```

```
        ↪ linkToItsFrequencyNumberConverter);
```

```
        var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
```

```
        ↪ sequenceToItsLocalElementLevelsConverter);
```

```
        var stringToUnicodeSequenceConverter = new
```

```
        ↪ StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
```

```
        ↪ index, optimalVariantConverter, unicodeSequenceMarker);
```

```
        var originalString = "Hello";
```

```
        var unicodeSequenceLink =
```

```
        ↪ stringToUnicodeSequenceConverter.Convert(originalString);
```

```
        var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
```

```
        ↪ unicodeSymbolMarker);
```

```
        var unicodeSymbolToCharConverter = new
```

```
        ↪ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
```

```
        ↪ unicodeSymbolCriterionMatcher);
```

```
        var unicodeSequenceCriterionMatcher = new TargetMatcher<ulong>(links,
```

```
        ↪ unicodeSequenceMarker);
```

```
        var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
```

```
        ↪ unicodeSymbolCriterionMatcher.IsMatched);
```

```
        var unicodeSequenceToStringConverter = new
```

```
        ↪ UnicodeSequenceToStringConverter<ulong>(links,
```

```
        ↪ unicodeSequenceCriterionMatcher, sequenceWalker,
```

```
        ↪ unicodeSymbolToCharConverter);
```

```
        var resultingString =
```

```
        ↪ unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
```

```
        Assert.Equal(originalString, resultingString);
```

```
    }
```

```
}
```

```
}
```

```
111 }
```

### 1.187 ./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     }
40 }
```

### 1.188 ./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     }
40 }
```

```
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, 234  
./csharp/Platform.Data.Doublets.Tests/ILinksExtensionsTests.cs, 235  
./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs, 235  
./csharp/Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs, 236  
./csharp/Platform.Data.Doublets.Tests/ReadSequenceTests.cs, 239  
./csharp/Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs, 240  
./csharp/Platform.Data.Doublets.Tests/ScopeTests.cs, 241  
./csharp/Platform.Data.Doublets.Tests/SequencesTests.cs, 241  
./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs, 256  
./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt32LinksTests.cs, 257  
./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt64LinksTests.cs, 257  
./csharp/Platform.Data.Doublets.Tests/TempLinksTestScope.cs, 258  
./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs, 259  
./csharp/Platform.Data.Doublets.Tests/UInt64LinksTests.cs, 262  
./csharp/Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs, 274  
./csharp/Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs, 275  
./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt32LinksTests.cs, 277  
./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt64LinksTests.cs, 277  
./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/ExternalLinksRecursionlessSizeBalancedTreeMethodsBase.cs, 32  
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSizeBalancedTreeMethodsBase.cs, 36  
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 39  
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSourcesSizeBalancedTreeMethods.cs, 40  
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 41  
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsSizeBalancedTreeMethods.cs, 42  
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksRecursionlessSizeBalancedTreeMethodsBase.cs, 43  
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSizeBalancedTreeMethodsBase.cs, 45  
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesLinkedListMethods.cs, 47  
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 49  
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesSizeBalancedTreeMethods.cs, 50  
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 51  
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsSizeBalancedTreeMethods.cs, 52  
./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinks.cs, 53  
./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinksBase.cs, 54  
./csharp/Platform.Data.Doublets/Memory/Split/Generic/UnusedLinksListMethods.cs, 65  
./csharp/Platform.Data.Doublets/Memory/Split/RawLinkDataPart.cs, 66  
./csharp/Platform.Data.Doublets/Memory/Split/RawLinkIndexPart.cs, 67

[illegible]



./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnusedLinksListMethods.cs, 145  
./csharp/Platform.Data.Doublets/Numbers/Raw/LongRawNumberSequenceToNumberConverter.cs, 145  
./csharp/Platform.Data.Doublets/Numbers/Raw/NumberToLongRawNumberSequenceConverter.cs, 146  
./csharp/Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs, 146  
./csharp/Platform.Data.Doublets/Numbers/Unary/LinkToFrequencyNumberConverter.cs, 147  
./csharp/Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs, 148  
./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs, 148  
./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs, 149  
./csharp/Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 150  
./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 151  
./csharp/Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 152  
./csharp/Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 153  
./csharp/Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 156  
./csharp/Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 156  
./csharp/Platform.Data.Doublets/Sequences/Converters/SequenceToLocalElementLevelsConverter.cs, 158  
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs, 158  
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs, 158  
./csharp/Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 159  
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 160  
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 160  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 162  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 164  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToFrequencyValueConverter.cs, 165  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 165  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 165  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 166  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 167  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 167  
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 167  
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 168  
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 169  
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 169  
./csharp/Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 169  
./csharp/Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 170  
./csharp/Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 171  
./csharp/Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 171  
./csharp/Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 172  
./csharp/Platform.Data.Doublets/Sequences/Indexes/Unindex.cs, 173  
./csharp/Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 173  
./csharp/Platform.Data.Doublets/Sequences/Sequences.cs, 200  
./csharp/Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 211  
./csharp/Platform.Data.Doublets/Sequences/SequencesOptions.cs, 211  
./csharp/Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 214  
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 214  
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 215  
./csharp/Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 217  
./csharp/Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 217  
./csharp/Platform.Data.Doublets/Stacks/Stack.cs, 218  
./csharp/Platform.Data.Doublets/Stacks/StackExtensions.cs, 219  
./csharp/Platform.Data.Doublets/SynchronizedLinks.cs, 219  
./csharp/Platform.Data.Doublets/Time/DateTimeToLongRawNumberSequenceConverter.cs, 220  
./csharp/Platform.Data.Doublets/Time/LongRawNumberSequenceToDateTimeConverter.cs, 220  
./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs, 221  
./csharp/Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 223  
./csharp/Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 228  
./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 229  
./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSymbolsListConverter.cs, 230  
./csharp/Platform.Data.Doublets/Unicode/UnicodeMap.cs, 230  
./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 233  
./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 233  
./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.cs, 234