

LinksPlatform's Platform.Data.Doublets Class Library

1.1 ./csharp/Platform.Data.Doublets/CriterionMatchers/TargetMatcher.cs

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

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

```
1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Decorators
6 {
7     public class LinksCascadeUniquenessAndUsagesResolver<TLink> : LinksUniquenessResolver<TLink>
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public LinksCascadeUniquenessAndUsagesResolver(ILinks<TLink> links) : base(links) { }
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         protected override TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
14             ↳ newLinkAddress)
15         {
16             // Use Facade (the last decorator) to ensure recursion working correctly
17             _facade.MergeUsages(oldLinkAddress, newLinkAddress);
18             return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
19         }
20     }
21 }
```

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

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

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Decorators
8  {
9      public abstract class LinksDecoratorBase<TLink> : LinksOperatorBase<TLink>, ILinks<TLink>
10     {
11         protected readonly LinksConstants<TLink> _constants;
12
13         public LinksConstants<TLink> Constants
14         {
15             [MethodImpl(MethodImplOptions.AggressiveInlining)]
16             get => _constants;
17         }
18
19         protected ILinks<TLink> _facade;
20
21         public ILinks<TLink> Facade
22         {
23             [MethodImpl(MethodImplOptions.AggressiveInlining)]
24             get => _facade;
25             [MethodImpl(MethodImplOptions.AggressiveInlining)]
26             set
27             {
28                 _facade = value;
29                 if (_links is LinksDecoratorBase<TLink> decorator)
30                 {
31                     decorator.Facade = value;
32                 }
33             }
34         }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected LinksDecoratorBase(ILinks<TLink> links) : base(links)
38         {
39             _constants = links.Constants;
40             Facade = this;
41         }
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         public virtual TLink Count(IList<TLink> restrictions) => _links.Count(restrictions);
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
48             => _links.Each(handler, restrictions);
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public virtual TLink Create(IList<TLink> restrictions) => _links.Create(restrictions);
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
55             _links.Update(restrictions, substitution);
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         public virtual void Delete(IList<TLink> restrictions) => _links.Delete(restrictions);
59     }
60 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Disposables;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5  #pragma warning disable CA1063 // Implement IDisposable Correctly
6
7  namespace Platform.Data.Doublets.Decorators
8  {
9      public abstract class LinksDisposableDecoratorBase<TLink> : LinksDecoratorBase<TLink>,
10         ↳ ILinks<TLink>, System.IDisposable
11     {
12         protected class DisposableWithMultipleCallsAllowed : Disposable
13         {
14             [MethodImpl(MethodImplOptions.AggressiveInlining)]
15             public DisposableWithMultipleCallsAllowed(Disposal disposal) : base(disposal) { }
16
17             protected override bool AllowMultipleDisposeCalls

```

```

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

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Decorators
8 {
9     // TODO: Make LinksExternalReferenceValidator. A layer that checks each link to exist or to
10     // be external (hybrid link's raw number).
11     public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
18         {
19             var links = _links;
20             links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
21             return links.Each(handler, restrictions);
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
26         {
27             // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
28             var links = _links;
29             links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
30             links.EnsureInnerReferenceExists(substitution, nameof(substitution));
31             return links.Update(restrictions, substitution);
32         }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public override void Delete(IList<TLink> restrictions)
36         {
37             var link = restrictions[_constants.IndexPart];
38             var links = _links;
39             links.EnsureLinkExists(link, nameof(link));
40             links.Delete(link);
41         }
42     }
43 }

```

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

```

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

```

```

5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Decorators
8  {
9      public class LinksItselfConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public LinksItselfConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
19         {
20             var constants = _constants;
21             var itselfConstant = constants.Itself;
22             if (!_equalityComparer.Equals(constants.Any, itselfConstant) &&
23                 ↪ restrictions.Contains(itselfConstant))
24             {
25                 // Itself constant is not supported for Each method right now, skipping execution
26                 return constants.Continue;
27             }
28             return _links.Each(handler, restrictions);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
32             ↪ _links.Update(restrictions, _links.ResolveConstantAsSelfReference(_constants.Itself,
33                 ↪ restrictions, substitution));
34     }
35 }

```

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

```

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

```

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

```

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

```

```

14         public override TLink Create(IList<TLink> restrictions) => _links.CreatePoint();
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
18             ↪ _links.Update(restrictions, _links.ResolveConstantAsSelfReference(_constants.Null,
19             ↪ restrictions, substitution));
18     }
19 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      public class LinksUniquenessResolver<TLink> : LinksDecoratorBase<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↪ EqualityComparer<TLink>.Default;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public LinksUniquenessResolver(ILinks<TLink> links) : base(links) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
18         {
19             var constants = _constants;
20             var links = _links;
21             var newLinkAddress = links.SearchOrDefault(substitution[constants.SourcePart],
22             ↪ substitution[constants.TargetPart]);
23             if (_equalityComparer.Equals(newLinkAddress, default))
24             {
25                 return links.Update(restrictions, substitution);
26             }
27             return ResolveAddressChangeConflict(restrictions[constants.IndexPart],
28             ↪ newLinkAddress);
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
33             ↪ newLinkAddress)
34         {
35             if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
36             ↪ _links.Exists(oldLinkAddress))
37             {
38                 _facade.Delete(oldLinkAddress);
39             }
40             return newLinkAddress;
41         }
42     }
43 }

```

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

```

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

```

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

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

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

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

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

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using TLink = System.UInt32;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Decorators
8 {
9     public class UInt32Links : LinksDisposableDecoratorBase<TLink>
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public UInt32Links(ILinks<TLink> links) : base(links) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public override TLink Create(IList<TLink> restrictions) => _links.CreatePoint();
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
19         {
20             var constants = _constants;
21             var indexPartConstant = constants.IndexPart;
22             var sourcePartConstant = constants.SourcePart;
23             var targetPartConstant = constants.TargetPart;
```

```

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

```

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

```

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

```

```

28 {
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public UInt64Links(ILinks<ulong> links) : base(links) { }
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     public override ulong Create(IList<ulong> restrictions) => _links.CreatePoint();
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
37     {
38         var constants = _constants;
39         var indexPartConstant = constants.IndexPart;
40         var sourcePartConstant = constants.SourcePart;
41         var targetPartConstant = constants.TargetPart;
42         var nullConstant = constants.Null;
43         var itselfConstant = constants.Itself;
44         var existedLink = nullConstant;
45         var updatedLink = restrictions[indexPartConstant];
46         var newSource = substitution[sourcePartConstant];
47         var newTarget = substitution[targetPartConstant];
48         var links = _links;
49         if (newSource != itselfConstant && newTarget != itselfConstant)
50         {
51             existedLink = links.SearchOrDefault(newSource, newTarget);
52         }
53         if (existedLink == nullConstant)
54         {
55             var before = links.GetLink(updatedLink);
56             if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
57                 ↪ newTarget)
58             {
59                 links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
60                     ↪ newSource,
61                                     newTarget == itselfConstant ? updatedLink :
62                     ↪ newTarget);
63             }
64             return updatedLink;
65         }
66         else
67         {
68             return _facade.MergeAndDelete(updatedLink, existedLink);
69         }
70     }
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     public override void Delete(IList<ulong> restrictions)
74     {
75         var linkIndex = restrictions[_constants.IndexPart];
76         var links = _links;
77         links.EnforceResetValues(linkIndex);
78         _facade.DeleteAllUsages(linkIndex);
79         links.Delete(linkIndex);
80     }
81 }
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     ///    return Constants.Break;
96     /// }
97     /// }
98     /// }
99     /// }
100     /// }

```

```

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                 //                 {
179                     //                     return Constants.Break;
180                     //                 }
181                 //                 if (Equals(substitutedDecision, Constants.Continue))
182                 //                 {
183                     //                     // Actual update here
184                     //                     Memory.SetLinkValue(newValue);
185                     //                 }
186                 //                 if (Equals(substitutedDecision, Constants.Skip))
187                 //                 {
188                     //                     // Cancel the update. TODO: decide use separate Cancel
189                     //                     // constant or Skip is enough?
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
10     /// <summary>
11     /// <para>.</para>
12     /// <para>.</para>
13     /// </summary>
14     /// <typeparam>
15     /// <para>.</para>
16     /// <para>.</para>
17     /// </typeparam>
18     public struct Doublet<T> : IEquatable<Doublet<T>>
19     {
20         private static readonly EqualityComparer<T> _equalityComparer =
            ↳ EqualityComparer<T>.Default;
21
22         /// <summary>
23         /// <para>.</para>
24         /// <para>.</para>
25         /// </summary>
26         /// <typeparam name="T">
27         /// <para>.</para>
28         /// <para>.</para>
29         /// </typeparam>
30         public readonly T Source;
31
32         /// <summary>
33         /// <para>.</para>
34         /// <para>.</para>
35         /// </summary>
36         /// <typeparam name="T">
37         /// <para>.</para>
38         /// <para>.</para>
39         /// </typeparam>
40         public readonly T Target;
41
42         /// <summary>
43         /// <para>.</para>
44         /// <para>.</para>
45         /// </summary>
46         /// <typeparam name="T">
47         /// <para>.</para>
48         /// <para>.</para>
49         /// </typeparam>
50         /// <param name="source">
51         /// <para>.</para>
52         /// <para>.</para>

```

```

53     /// </param>
54     /// <param name="target">
55     /// <para>.</para>
56     /// <para>.</para>
57     /// </param>
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     public Doublet(T source, T target)
60     {
61         Source = source;
62         Target = target;
63     }
64
65     /// <summary>
66     /// <para>.</para>
67     /// <para>.</para>
68     /// </summary>
69     /// <returns>
70     /// <para>.</para>
71     /// <para>.</para>
72     /// </returns>
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     public override string ToString() => $"{Source}->{Target}";
75
76     /// <summary>
77     /// <para>.</para>
78     /// <para>.</para>
79     /// </summary>
80     /// <typeparam>
81     /// <para>.</para>
82     /// <para>.</para>
83     /// </typeparam>
84     /// <param name="other">
85     /// <para>.</para>
86     /// <para>.</para>
87     /// </param>
88     /// <returns>
89     /// <para>.</para>
90     /// <para>.</para>
91     /// </returns>
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
94     ↪ && _equalityComparer.Equals(Target, other.Target);
95
96     /// <summary>
97     /// <para>.</para>
98     /// <para>.</para>
99     /// </summary>
100    /// <typeparam>
101    /// <para>.</para>
102    /// <para>.</para>
103    /// </typeparam>
104    /// <param name="obj">
105    /// <para>.</para>
106    /// <para>.</para>
107    /// </param>
108    /// <returns>
109    /// <para>.</para>
110    /// <para>.</para>
111    /// </returns>
112    [MethodImpl(MethodImplOptions.AggressiveInlining)]
113    public override bool Equals(object obj) => obj is Doublet<T> doublet ?
114    ↪ base.Equals(doublet) : false;
115
116    /// <summary>
117    /// <para>.</para>
118    /// <para>.</para>
119    /// </summary>
120    /// <returns>
121    /// <para>.</para>
122    /// <para>.</para>
123    /// </returns>
124    [MethodImpl(MethodImplOptions.AggressiveInlining)]
125    public override int GetHashCode() => (Source, Target).GetHashCode();
126
127    /// <summary>
128    /// <para>.</para>
129    /// <para>.</para>
130    /// </summary>

```

```

129     /// <param name="left">
130     /// <para>.</para>
131     /// <para>.</para>
132     /// </param>
133     /// <param name="right">
134     /// <para>.</para>
135     /// <para>.</para>
136     /// </param>
137     /// <returns>
138     /// <para>.</para>
139     /// <para>.</para>
140     /// </returns>
141     [MethodImpl(MethodImplOptions.AggressiveInlining)]
142     public static bool operator ==(Doublet<T> left, Doublet<T> right) => left.Equals(right);
143
144     /// <summary>
145     /// <para>.</para>
146     /// <para>.</para>
147     /// </summary>
148     /// <param name="left">
149     /// <para>.</para>
150     /// <para>.</para>
151     /// </param>
152     /// <param name="right">
153     /// <para>.</para>
154     /// <para>.</para>
155     /// </param>
156     /// <returns>
157     /// <para>.</para>
158     /// <para>.</para>
159     /// </returns>
160     [MethodImpl(MethodImplOptions.AggressiveInlining)]
161     public static bool operator !=(Doublet<T> left, Doublet<T> right) => !(left == right);
162 }
163 }

```

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 = OUL; 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 = OUL; i < amountOfSearches; i++)
48             {
49                 var linksAddressRange = new Range<ulong>(0,
50                     ↳ addressToUInt64Converter.Convert(links.Count()));
51                 var source =
52                     ↳ uint64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
53                 var target =
54                     ↳ uint64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
55                 links.SearchOrDefault(source, target);
56             }
57         }
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, ulong
61             ↳ amountOfDeletions)
62         {
63             var random = RandomHelpers.Default;
64             var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
65             var uint64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
66             var linksCount = addressToUInt64Converter.Convert(links.Count());
67             var min = amountOfDeletions > linksCount ? OUL : linksCount - amountOfDeletions;
68             for (var i = OUL; i < amountOfDeletions; i++)
69             {
70                 linksCount = addressToUInt64Converter.Convert(links.Count());
71                 if (linksCount <= min)
72                 {
73                     break;
74                 }
75                 var linksAddressRange = new Range<ulong>(min, linksCount);
76                 var link =
77                     ↳ uint64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
78                 links.Delete(link);
79             }
80         }
81
82         [MethodImpl(MethodImplOptions.AggressiveInlining)]
83         public static void Delete<TLink>(this ILinks<TLink> links, TLink linkToDelete) =>
84             ↳ links.Delete(new LinkAddress<TLink>(linkToDelete));
85     }
86 }

```



```

75  /// <remarks>
76  /// TODO: Возможно есть очень простой способ это сделать.
77  /// (Например просто удалить файл, или изменить его размер таким образом,
78  /// чтобы удалился весь контент)
79  /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
80  /// </remarks>
81  [MethodImpl(MethodImplOptions.AggressiveInlining)]
82  public static void DeleteAll<TLink>(this ILinks<TLink> links)
83  {
84      var equalityComparer = EqualityComparer<TLink>.Default;
85      var comparer = Comparer<TLink>.Default;
86      for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
87          ↪ Arithmetic.Decrement(i))
88      {
89          links.Delete(i);
90          if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
91          {
92              i = links.Count();
93          }
94      }
95  }
96  [MethodImpl(MethodImplOptions.AggressiveInlining)]
97  public static TLink First<TLink>(this ILinks<TLink> links)
98  {
99      TLink firstLink = default;
100     var equalityComparer = EqualityComparer<TLink>.Default;
101     if (equalityComparer.Equals(links.Count(), default))
102     {
103         throw new InvalidOperationException("В хранилище нет связей.");
104     }
105     links.Each(links.Constants.Any, links.Constants.Any, link =>
106     {
107         firstLink = link[links.Constants.IndexPart];
108         return links.Constants.Break;
109     });
110     if (equalityComparer.Equals(firstLink, default))
111     {
112         throw new InvalidOperationException("В процессе поиска по хранилищу не было
113             ↪ найдено связей.");
114     }
115     return firstLink;
116 }
117 [MethodImpl(MethodImplOptions.AggressiveInlining)]
118 public static IList<TLink> SingleOrDefault<TLink>(this ILinks<TLink> links, IList<TLink>
119     ↪ query)
120 {
121     IList<TLink> result = null;
122     var count = 0;
123     var constants = links.Constants;
124     var @continue = constants.Continue;
125     var @break = constants.Break;
126     links.Each(linkHandler, query);
127     return result;
128 }
129 TLink linkHandler(IList<TLink> link)
130 {
131     if (count == 0)
132     {
133         result = link;
134         count++;
135         return @continue;
136     }
137     else
138     {
139         result = null;
140         return @break;
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

```

```

150 /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
151 /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
152 /// </remarks>
153 [MethodImpl(MethodImplOptions.AggressiveInlining)]
154 public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
    ↪ path)
155 {
156     var current = path[0];
157     //EnsureLinkExists(current, "path");
158     if (!links.Exists(current))
159     {
160         return false;
161     }
162     var equalityComparer = EqualityComparer<TLink>.Default;
163     var constants = links.Constants;
164     for (var i = 1; i < path.Length; i++)
165     {
166         var next = path[i];
167         var values = links.GetLink(current);
168         var source = values[constants.SourcePart];
169         var target = values[constants.TargetPart];
170         if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
            ↪ next))
171         {
172             //throw new InvalidOperationException(string.Format("Невозможно выбрать
            ↪ путь, так как и Source и Target совпадают с элементом пути {0}.", next));
173             return false;
174         }
175         if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
            ↪ target))
176         {
177             //throw new InvalidOperationException(string.Format("Невозможно продолжить
            ↪ путь через элемент пути {0}", next));
178             return false;
179         }
180         current = next;
181     }
182     return true;
183 }
184
185 /// <remarks>
186 /// Может потребовать дополнительного стека для PathElement's при использовании
    ↪ SequenceWalker.
187 /// </remarks>
188 [MethodImpl(MethodImplOptions.AggressiveInlining)]
189 public static TLink GetByKeyes<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;

```

```

}
#endregion

/// <summary>
/// Возвращает индекс указанной связи.
/// </summary>
/// <param name="links">Хранилище связей.</param>
/// <param name="link">Связь представленная списком, состоящим из её адреса и
  ↳ содержимого.</param>
/// <returns>Индекс начальной связи для указанной связи.</returns>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
  ↳ link[links.Constants.IndexPart];

/// <summary>
/// Возвращает индекс начальной (Source) связи для указанной связи.
/// </summary>
/// <param name="links">Хранилище связей.</param>
/// <param name="link">Индекс связи.</param>
/// <returns>Индекс начальной связи для указанной связи.</returns>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
  ↳ links.GetLink(link)[links.Constants.SourcePart];

/// <summary>
/// Возвращает индекс начальной (Source) связи для указанной связи.
/// </summary>
/// <param name="links">Хранилище связей.</param>
/// <param name="link">Связь представленная списком, состоящим из её адреса и
  ↳ содержимого.</param>
/// <returns>Индекс начальной связи для указанной связи.</returns>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
  ↳ link[links.Constants.SourcePart];

/// <summary>
/// Возвращает индекс конечной (Target) связи для указанной связи.
/// </summary>
/// <param name="links">Хранилище связей.</param>
/// <param name="link">Индекс связи.</param>
/// <returns>Индекс конечной связи для указанной связи.</returns>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
  ↳ links.GetLink(link)[links.Constants.TargetPart];

/// <summary>
/// Возвращает индекс конечной (Target) связи для указанной связи.
/// </summary>
/// <param name="links">Хранилище связей.</param>
/// <param name="link">Связь представленная списком, состоящим из её адреса и
  ↳ содержимого.</param>
/// <returns>Индекс конечной связи для указанной связи.</returns>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
  ↳ link[links.Constants.TargetPart];

/// <summary>
/// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
  ↳ (handler) для каждой подходящей связи.
/// </summary>
/// <param name="links">Хранилище связей.</param>
/// <param name="handler">Обработчик каждой подходящей связи.</param>
/// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
  ↳ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
  ↳ Any - отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
/// <returns>True, в случае если проход по связям не был прерван и False в обратном
  ↳ случае.</returns>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
  ↳ handler, params TLink[] restrictions)
  => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
    ↳ links.Constants.Continue);

/// <summary>
/// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
  ↳ (handler) для каждой подходящей связи.

```

```

281 /// </summary>
282 /// <param name="links">Хранилище связей.</param>
283 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
    ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
284 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
    ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
285 /// <param name="handler">Обработчик каждой подходящей связи.</param>
286 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
287 [MethodImpl(MethodImplOptions.AggressiveInlining)]
288 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
    ↳ Func<TLink, bool> handler)
289 {
290     var constants = links.Constants;
291     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
    ↳ constants.Break, constants.Any, source, target);
292 }
293
294 /// <summary>
295 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
296 /// </summary>
297 /// <param name="links">Хранилище связей.</param>
298 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
    ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
299 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
    ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
300 /// <param name="handler">Обработчик каждой подходящей связи.</param>
301 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
302 [MethodImpl(MethodImplOptions.AggressiveInlining)]
303 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
    ↳ Func<IList<TLink>, TLink> handler) => links.Each(handler, links.Constants.Any,
    ↳ source, target);
304
305 [MethodImpl(MethodImplOptions.AggressiveInlining)]
306 public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ 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
341   ↳ в хранилище связей.
342 /// </summary>
343 /// <param name="links">Хранилище связей.</param>
344 /// <param name="source">Начало связи.</param>
345 /// <param name="target">Конец связи.</param>
346 /// <returns>Значение, определяющее существует ли связь.</returns>
347 [MethodImpl(MethodImplOptions.AggressiveInlining)]
348 public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
349   ↳ => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
350   ↳ default) > 0;
351
352 #region Ensure
353 // TODO: May be move to EnsureExtensions or make it both there and here
354
355 [MethodImpl(MethodImplOptions.AggressiveInlining)]
356 public static void EnsureLinkExists<TLink>(this ILinks<TLink> links, IList<TLink>
357   ↳ restrictions)
358 {
359     for (var i = 0; i < restrictions.Count; i++)
360     {
361         if (!links.Exists(restrictions[i]))
362         {
363             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
364               ↳ $"sequence[{i}]");
365         }
366     }
367 }
368
369 [MethodImpl(MethodImplOptions.AggressiveInlining)]
370 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
371   ↳ reference, string argumentName)
372 {
373     if (links.Constants.IsInternalReference(reference) && !links.Exists(reference))
374     {
375         throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
376     }
377 }
378
379 [MethodImpl(MethodImplOptions.AggressiveInlining)]
380 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
381   ↳ IList<TLink> restrictions, string argumentName)
382 {
383     for (int i = 0; i < restrictions.Count; i++)
384     {
385         links.EnsureInnerReferenceExists(restrictions[i], argumentName);
386     }
387 }
388
389 [MethodImpl(MethodImplOptions.AggressiveInlining)]
390 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
391   ↳ restrictions)
392 {
393     var equalityComparer = EqualityComparer<TLink>.Default;
394     var any = links.Constants.Any;
395     for (var i = 0; i < restrictions.Count; i++)
396     {
397         if (!equalityComparer.Equals(restrictions[i], any) &&
398           ↳ !links.Exists(restrictions[i]))
399         {
400             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
401               ↳ $"sequence[{i}]");
402         }
403     }
404 }
405
406 [MethodImpl(MethodImplOptions.AggressiveInlining)]
407 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
408   ↳ string argumentName)
409 {
410     var equalityComparer = EqualityComparer<TLink>.Default;
411     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
412     {
413         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
414     }
415 }

```

```

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,
481         ↪ constants.Any));
482     var equalityComparer = EqualityComparer<TLink>.Default;
483     if (equalityComparer.Equals(values[constants.SourcePart], link))
484     {
485         usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
486     }
487     TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
488         ↪ link));
489     if (equalityComparer.Equals(values[constants.TargetPart], link))
490     {
491         usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
492     }
493     return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
494 }
495
496 /// <param name="links">Хранилище связей.</param>
497 [MethodImpl(MethodImplOptions.AggressiveInlining)]
498 public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
499     ↪ Comparer<TLink>.Default.Compare(links.CountUsages(link), default) > 0;
500
501 /// <param name="links">Хранилище связей.</param>
502 [MethodImpl(MethodImplOptions.AggressiveInlining)]
503 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
504     ↪ TLink target)
505 {
506     var constants = links.Constants;
507     var values = links.GetLink(link);
508     var equalityComparer = EqualityComparer<TLink>.Default;
509     return equalityComparer.Equals(values[constants.SourcePart], source) &&
510         ↪ equalityComparer.Equals(values[constants.TargetPart], target);
511 }
512
513 /// <summary>
514 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
515 /// </summary>
516 /// <param name="links">Хранилище связей.</param>
517 /// <param name="source">Индекс связи, которая является началом для искомой
518     ↪ связи.</param>
519 /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
520 /// <returns>Индекс искомой связи с указанными Source (началом) и Target
521     ↪ (концом).</returns>
522 [MethodImpl(MethodImplOptions.AggressiveInlining)]
523 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
524     ↪ target)
525 {
526     var constants = links.Constants;
527     var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break, default);
528     links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
529     return setter.Result;
530 }
531
532 /// <param name="links">Хранилище связей.</param>
533 [MethodImpl(MethodImplOptions.AggressiveInlining)]
534 public static TLink Create<TLink>(this ILinks<TLink> links) => links.Create(null);
535
536 /// <param name="links">Хранилище связей.</param>
537 [MethodImpl(MethodImplOptions.AggressiveInlining)]
538 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
539 {
540     var link = links.Create();
541     return links.Update(link, link, link);
542 }
543
544 /// <param name="links">Хранилище связей.</param>
545 [MethodImpl(MethodImplOptions.AggressiveInlining)]
546 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
547     ↪ target) => links.Update(links.Create(), source, target);
548
549 /// <summary>
550 /// 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
    ↳ выполняется обновление.</param>
547 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    ↳ выполняется обновление.</param>
548 /// <returns>Индекс обновлённой связи.</returns>
549 [MethodImpl(MethodImplOptions.AggressiveInlining)]
550 public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
    ↳ TLink newTarget) => links.Update(new LinkAddress<TLink>(link), new Link<TLink>(link,
    ↳ newSource, newTarget));
551
552 /// <summary>
553 /// Обновляет связь с указанными началом (Source) и концом (Target)
554 /// на связь с указанными началом (NewSource) и концом (NewTarget).
555 /// </summary>
556 /// <param name="links">Хранилище связей.</param>
557 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
    ↳ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
    ↳ Itself - требование установить ссылку на себя, 1.. $\infty$  конкретный адрес другой
    ↳ связи.</param>
558 /// <returns>Индекс обновлённой связи.</returns>
559 [MethodImpl(MethodImplOptions.AggressiveInlining)]
560 public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
561 {
562     if (restrictions.Length == 2)
563     {
564         return links.MergeAndDelete(restrictions[0], restrictions[1]);
565     }
566     if (restrictions.Length == 4)
567     {
568         return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
    ↳ restrictions[2], restrictions[3]);
569     }
570     else
571     {
572         return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
573     }
574 }
575
576 [MethodImpl(MethodImplOptions.AggressiveInlining)]
577 public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
    ↳ links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
578 {
579     var equalityComparer = EqualityComparer<TLink>.Default;
580     var constants = links.Constants;
581     var restrictionsIndex = restrictions[constants.IndexPart];
582     var substitutionIndex = substitution[constants.IndexPart];
583     if (equalityComparer.Equals(substitutionIndex, default))
584     {
585         substitutionIndex = restrictionsIndex;
586     }
587     var source = substitution[constants.SourcePart];
588     var target = substitution[constants.TargetPart];
589     source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
590     target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
591     return new Link<TLink>(substitutionIndex, source, target);
592 }
593
594 /// <summary>
595 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
    ↳ с указанными Source (началом) и Target (концом).
596 /// </summary>
597 /// <param name="links">Хранилище связей.</param>
598 /// <param name="source">Индекс связи, которая является началом на создаваемой
    ↳ связи.</param>
599 /// <param name="target">Индекс связи, которая является концом для создаваемой
    ↳ связи.</param>
600 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
601 [MethodImpl(MethodImplOptions.AggressiveInlining)]
602 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target)
603 {
604     var link = links.SearchOrDefault(source, target);
605     if (EqualityComparer<TLink>.Default.Equals(link, default))

```



```

606     {
607         link = links.CreateAndUpdate(source, target);
608     }
609     return link;
610 }
611
612 /// <summary>
613 /// Обновляет связь с указанными началом (Source) и концом (Target)
614 /// на связь с указанными началом (NewSource) и концом (NewTarget).
615 /// </summary>
616 /// <param name="links">Хранилище связей.</param>
617 /// <param name="source">Индекс связи, которая является началом обновляемой
    → связи.</param>
618 /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
619 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    → выполняется обновление.</param>
620 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    → выполняется обновление.</param>
621 /// <returns>Индекс обновлённой связи.</returns>
622 [MethodImpl(MethodImplOptions.AggressiveInlining)]
623 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
    → TLink target, TLink newSource, TLink newTarget)
624 {
625     var equalityComparer = EqualityComparer<TLink>.Default;
626     var link = links.SearchOrDefault(source, target);
627     if (equalityComparer.Equals(link, default))
628     {
629         return links.CreateAndUpdate(newSource, newTarget);
630     }
631     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
    → target))
632     {
633         return link;
634     }
635     return links.Update(link, newSource, newTarget);
636 }
637
638 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
639 /// <param name="links">Хранилище связей.</param>
640 /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
641 /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
642 [MethodImpl(MethodImplOptions.AggressiveInlining)]
643 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
    → target)
644 {
645     var link = links.SearchOrDefault(source, target);
646     if (!EqualityComparer<TLink>.Default.Equals(link, default))
647     {
648         links.Delete(link);
649         return link;
650     }
651     return default;
652 }
653
654 /// <summary>Удаляет несколько связей.</summary>
655 /// <param name="links">Хранилище связей.</param>
656 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
657 [MethodImpl(MethodImplOptions.AggressiveInlining)]
658 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
659 {
660     for (int i = 0; i < deletedLinks.Count; i++)
661     {
662         links.Delete(deletedLinks[i]);
663     }
664 }
665
666 /// <remarks>Before execution of this method ensure that deleted link is detached (all
    → values - source and target are reset to null) or it might enter into infinite
    → recursion.</remarks>
667 [MethodImpl(MethodImplOptions.AggressiveInlining)]
668 public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
669 {
670     var anyConstant = links.Constants.Any;
671     var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
672     links.DeleteByQuery(usagesAsSourceQuery);
673     var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
674     links.DeleteByQuery(usagesAsTargetQuery);
675 }

```

```

676 [MethodImpl(MethodImplOptions.AggressiveInlining)]
677 public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
678 {
679     var count = CheckedConverter<TLink, long>.Default.Convert(links.Count(query));
680     if (count > 0)
681     {
682         var queryResult = new TLink[count];
683         var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
684             ↪ links.Constants.Continue);
685         links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
686         for (var i = count - 1; i >= 0; i--)
687         {
688             links.Delete(queryResult[i]);
689         }
690     }
691 }
692
693 // TODO: Move to Platform.Data
694 [MethodImpl(MethodImplOptions.AggressiveInlining)]
695 public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
696 {
697     var nullConstant = links.Constants.Null;
698     var equalityComparer = EqualityComparer<TLink>.Default;
699     var link = links.GetLink(linkIndex);
700     for (int i = 1; i < link.Count; i++)
701     {
702         if (!equalityComparer.Equals(link[i], nullConstant))
703         {
704             return false;
705         }
706     }
707     return true;
708 }
709
710 // TODO: Create a universal version of this method in Platform.Data (with using of for
711 ↪ loop)
712 [MethodImpl(MethodImplOptions.AggressiveInlining)]
713 public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
714 {
715     var nullConstant = links.Constants.Null;
716     var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
717     links.Update(updateRequest);
718 }
719
720 // TODO: Create a universal version of this method in Platform.Data (with using of for
721 ↪ loop)
722 [MethodImpl(MethodImplOptions.AggressiveInlining)]
723 public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
724 {
725     if (!links.AreValuesReset(linkIndex))
726     {
727         links.ResetValues(linkIndex);
728     }
729 }
730
731 /// <summary>
732 /// Merging two usages graphs, all children of old link moved to be children of new link
733 ↪ or deleted.
734 /// </summary>
735 [MethodImpl(MethodImplOptions.AggressiveInlining)]
736 public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
737     ↪ TLink newLinkIndex)
738 {
739     var addressToInt64Converter = CheckedConverter<TLink, long>.Default;
740     var equalityComparer = EqualityComparer<TLink>.Default;
741     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
742     {
743         var constants = links.Constants;
744         var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
745             ↪ constants.Any);
746         var usagesAsSourceCount =
747             ↪ addressToInt64Converter.Convert(links.Count(usagesAsSourceQuery));
748         var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
749             ↪ oldLinkIndex);
750         var usagesAsTargetCount =
751             ↪ addressToInt64Converter.Convert(links.Count(usagesAsTargetQuery));

```

```

744     var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
745         ↳ usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
746     if (!isStandalonePoint)
747     {
748         var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
749         if (totalUsages > 0)
750         {
751             var usages = ArrayPool.Allocate<TLink>(totalUsages);
752             var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
753                 ↳ links.Constants.Continue);
754             var i = 0L;
755             if (usagesAsSourceCount > 0)
756             {
757                 links.Each(usagesFiller.AddFirstAndReturnConstant,
758                     ↳ usagesAsSourceQuery);
759                 for (; i < usagesAsSourceCount; i++)
760                 {
761                     var usage = usages[i];
762                     if (!equalityComparer.Equals(usage, oldLinkIndex))
763                     {
764                         links.Update(usage, newLinkIndex, links.GetTarget(usage));
765                     }
766                 }
767             }
768             if (usagesAsTargetCount > 0)
769             {
770                 links.Each(usagesFiller.AddFirstAndReturnConstant,
771                     ↳ usagesAsTargetQuery);
772                 for (; i < usages.Length; i++)
773                 {
774                     var usage = usages[i];
775                     if (!equalityComparer.Equals(usage, oldLinkIndex))
776                     {
777                         links.Update(usage, links.GetSource(usage), newLinkIndex);
778                     }
779                 }
780             }
781             ArrayPool.Free(usages);
782         }
783     }
784     return newLinkIndex;
785 }
786
787 /// <summary>
788 /// Replace one link with another (replaced link is deleted, children are updated or
789   ↳ deleted).
790 /// </summary>
791 [MethodImpl(MethodImplOptions.AggressiveInlining)]
792 public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
793     ↳ TLink newLinkIndex)
794 {
795     var equalityComparer = EqualityComparer<TLink>.Default;
796     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
797     {
798         links.MergeUsages(oldLinkIndex, newLinkIndex);
799         links.Delete(oldLinkIndex);
800     }
801     return newLinkIndex;
802 }
803
804 [MethodImpl(MethodImplOptions.AggressiveInlining)]
805 public static ILinks<TLink>
806     ↳ DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
807 {
808     links = new LinksCascadeUsagesResolver<TLink>(links);
809     links = new NonNullContentsLinkDeletionResolver<TLink>(links);
810     links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
811     return links;
812 }
813
814 [MethodImpl(MethodImplOptions.AggressiveInlining)]
815 public static string Format<TLink>(this ILinks<TLink> links, IList<TLink> link)
816 {
817     var constants = links.Constants;
818     return $"({link[constants.IndexPart]}: {link[constants.SourcePart]}
819         ↳ {link[constants.TargetPart]});";
820 }

```

```

814     [MethodImpl(MethodImplOptions.AggressiveInlining)]
815     public static string Format<TLink>(this ILinks<TLink> links, TLink link) =>
816         ↪ 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>,
6          ↪ LinksConstants<TLink>>, ILinks<TLink>
7      {
8      }
9  }

```

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

```

1  using Platform.Collections.Lists;
2  using Platform.Exceptions;
3  using Platform.Ranges;
4  using Platform.Singletons;
5  using System;
6  using System.Collections;
7  using System.Collections.Generic;
8  using System.Runtime.CompilerServices;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets
13 {
14     /// <summary>
15     /// Структура описывающая уникальную связь.
16     /// </summary>
17     public struct Link<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>, IList<TLink>
18     {
19         public static readonly Link<TLink> Null = new Link<TLink>();
20
21         private static readonly LinksConstants<TLink> _constants =
22             ↪ Default<LinksConstants<TLink>>.Instance;
23         private static readonly EqualityComparer<TLink> _equalityComparer =
24             ↪ EqualityComparer<TLink>.Default;
25
26         private const int Length = 3;
27
28         public readonly TLink Index;
29         public readonly TLink Source;
30         public readonly TLink Target;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public Link(params TLink[] values) => SetValues(values, out Index, out Source, out
34             ↪ Target);
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public Link(IList<TLink> values) => SetValues(values, out Index, out Source, out Target);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public Link(object other)
41         {
42             if (other is Link<TLink> otherLink)
43             {
44                 SetValues(ref otherLink, out Index, out Source, out Target);
45             }
46             else if (other is IList<TLink> otherList)
47             {
48                 SetValues(otherList, out Index, out Source, out Target);
49             }
50             else
51             {
52                 throw new NotSupportedException();
53             }
54         }
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         public Link(ref Link<TLink> other) => SetValues(ref other, out Index, out Source, out
58             ↪ Target);
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         public Link(TLink index, TLink source, TLink target)

```

```

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,
66     ↪ out TLink target)
67     {
68         index = other.Index;
69         source = other.Source;
70         target = other.Target;
71     }
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     private static void SetValues(IList<TLink> values, out TLink index, out TLink source,
75     ↪ out TLink target)
76     {
77         switch (values.Count)
78         {
79             case 3:
80                 index = values[0];
81                 source = values[1];
82                 target = values[2];
83                 break;
84             case 2:
85                 index = values[0];
86                 source = values[1];
87                 target = default;
88                 break;
89             case 1:
90                 index = values[0];
91                 source = default;
92                 target = default;
93                 break;
94             default:
95                 index = default;
96                 source = default;
97                 target = default;
98                 break;
99         }
100     }
101
102     [MethodImpl(MethodImplOptions.AggressiveInlining)]
103     public override int GetHashCode() => (Index, Source, Target).GetHashCode();
104
105     [MethodImpl(MethodImplOptions.AggressiveInlining)]
106     public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
107     && _equalityComparer.Equals(Source, _constants.Null)
108     && _equalityComparer.Equals(Target, _constants.Null);
109
110     [MethodImpl(MethodImplOptions.AggressiveInlining)]
111     public override bool Equals(object other) => other is Link<TLink> &&
112     ↪ Equals((Link<TLink>)other);
113
114     [MethodImpl(MethodImplOptions.AggressiveInlining)]
115     public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
116     && _equalityComparer.Equals(Source, other.Source)
117     && _equalityComparer.Equals(Target, other.Target);
118
119     [MethodImpl(MethodImplOptions.AggressiveInlining)]
120     public static string ToString(TLink index, TLink source, TLink target) => $"({index}:
121     ↪ {source}->{target})";
122
123     [MethodImpl(MethodImplOptions.AggressiveInlining)]
124     public static string ToString(TLink source, TLink target) => $"({source}->{target})";
125
126     [MethodImpl(MethodImplOptions.AggressiveInlining)]
127     public static implicit operator TLink[] (Link<TLink> link) => link.ToArray();
128
129     [MethodImpl(MethodImplOptions.AggressiveInlining)]
130     public static implicit operator Link<TLink> (TLink[] linkArray) => new
131     ↪ Link<TLink>(linkArray);
132
133     [MethodImpl(MethodImplOptions.AggressiveInlining)]
134     public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
135     ↪ ToString(Source, Target) : ToString(Index, Source, Target);
136
137     #region IList

```

```

132 public int Count
133 {
134     [MethodImpl(MethodImplOptions.AggressiveInlining)]
135     get => Length;
136 }
137
138 public bool IsReadOnly
139 {
140     [MethodImpl(MethodImplOptions.AggressiveInlining)]
141     get => true;
142 }
143
144 public TLink this[int index]
145 {
146     [MethodImpl(MethodImplOptions.AggressiveInlining)]
147     get
148     {
149         Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
150             ↪ nameof(index));
151         if (index == _constants.IndexPart)
152         {
153             return Index;
154         }
155         if (index == _constants.SourcePart)
156         {
157             return Source;
158         }
159         if (index == _constants.TargetPart)
160         {
161             return Target;
162         }
163         throw new NotSupportedException(); // Impossible path due to
164             ↪ Ensure.ArgumentInRange
165     }
166     [MethodImpl(MethodImplOptions.AggressiveInlining)]
167     set => throw new NotSupportedException();
168 }
169
170 [MethodImpl(MethodImplOptions.AggressiveInlining)]
171 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
172
173 [MethodImpl(MethodImplOptions.AggressiveInlining)]
174 public IEnumerator<TLink> GetEnumerator()
175 {
176     yield return Index;
177     yield return Source;
178     yield return Target;
179 }
180
181 [MethodImpl(MethodImplOptions.AggressiveInlining)]
182 public void Add(TLink item) => throw new NotSupportedException();
183
184 [MethodImpl(MethodImplOptions.AggressiveInlining)]
185 public void Clear() => throw new NotSupportedException();
186
187 [MethodImpl(MethodImplOptions.AggressiveInlining)]
188 public bool Contains(TLink item) => IndexOf(item) >= 0;
189
190 [MethodImpl(MethodImplOptions.AggressiveInlining)]
191 public void CopyTo(TLink[] array, int arrayIndex)
192 {
193     Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
194     Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
195         ↪ nameof(arrayIndex));
196     if (arrayIndex + Length > array.Length)
197     {
198         throw new InvalidOperationException();
199     }
200     array[arrayIndex++] = Index;
201     array[arrayIndex++] = Source;
202     array[arrayIndex] = Target;
203 }
204
205 [MethodImpl(MethodImplOptions.AggressiveInlining)]
206 public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
207
208 [MethodImpl(MethodImplOptions.AggressiveInlining)]
209 public int IndexOf(TLink item)

```

```

208     {
209         if (_equalityComparer.Equals(Index, item))
210         {
211             return _constants.IndexPart;
212         }
213         if (_equalityComparer.Equals(Source, item))
214         {
215             return _constants.SourcePart;
216         }
217         if (_equalityComparer.Equals(Target, item))
218         {
219             return _constants.TargetPart;
220         }
221         return -1;
222     }
223
224     [MethodImpl(MethodImplOptions.AggressiveInlining)]
225     public void Insert(int index, TLink item) => throw new NotSupportedException();
226
227     [MethodImpl(MethodImplOptions.AggressiveInlining)]
228     public void RemoveAt(int index) => throw new NotSupportedException();
229
230     [MethodImpl(MethodImplOptions.AggressiveInlining)]
231     public static bool operator ==(Link<TLink> left, Link<TLink> right) =>
232         left.Equals(right);
233
234     [MethodImpl(MethodImplOptions.AggressiveInlining)]
235     public static bool operator !=(Link<TLink> left, Link<TLink> right) => !(left == right);
236
237     #endregion
238 }

```

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

```

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

```

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

```

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

```

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

```

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

```

```

5 namespace Platform.Data.Doublets.Memory
6 {
7     public interface ILinksListMethods<TLink>
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        void Detach(TLink freeLink);
11
12        [MethodImpl(MethodImplOptions.AggressiveInlining)]
13        void AttachAsFirst(TLink link);
14    }
15 }

```

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

```

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

```

1.27 ./csharp/Platform.Data.Doublets/Memory/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 }

```

1.28 ./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 }

```



```

25 [MethodImpl(MethodImplOptions.AggressiveInlining)]
26 public override bool Equals(object obj) => obj is LinksHeader<TLink> linksHeader ?
    ↳ Equals(linksHeader) : false;
27
28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 public bool Equals(LinksHeader<TLink> other)
30     => _equalityComparer.Equals(AllocatedLinks, other.AllocatedLinks)
31     && _equalityComparer.Equals(ReservedLinks, other.ReservedLinks)
32     && _equalityComparer.Equals(FreeLinks, other.FreeLinks)
33     && _equalityComparer.Equals(FirstFreeLink, other.FirstFreeLink)
34     && _equalityComparer.Equals(RootAsSource, other.RootAsSource)
35     && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
36     && _equalityComparer.Equals>LastFreeLink, other.LastFreeLink)
37     && _equalityComparer.Equals(Reserved8, other.Reserved8);
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 public override int GetHashCode() => (AllocatedLinks, ReservedLinks, FreeLinks,
    ↳ FirstFreeLink, RootAsSource, RootAsTarget, LastFreeLink, Reserved8).GetHashCode();
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 public static bool operator ==(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
    ↳ left.Equals(right);
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 public static bool operator !=(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
    ↳ !(left == right);
47 }
48 }

```

1.29 ./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)]

```

```

46     protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
47         ↳ AsRef<LinksHeader<TLink>>(Header);
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
51         ↳ AsRef<RawLinkDataPart<TLink>>(LinksDataParts + (RawLinkDataPart<TLink>.SizeInBytes *
52         ↳ _addressToInt64Converter.Convert(link)));
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
56         ↳ ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
57         ↳ (RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link)));
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
61     {
62         ref var link = ref GetLinkDataPartReference(linkIndex);
63         return new Link<TLink>(linkIndex, link.Source, link.Target);
64     }
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override bool FirstIsToLeftOfSecond(TLink first, TLink second)
68     {
69         ref var firstLink = ref GetLinkDataPartReference(first);
70         ref var secondLink = ref GetLinkDataPartReference(second);
71         return FirstIsToLeftOfSecond(firstLink.Source, firstLink.Target,
72         ↳ secondLink.Source, secondLink.Target);
73     }
74
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
77     {
78         ref var firstLink = ref GetLinkDataPartReference(first);
79         ref var secondLink = ref GetLinkDataPartReference(second);
80         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
81         ↳ secondLink.Source, secondLink.Target);
82     }
83
84     public TLink this[TLink index]
85     {
86         [MethodImpl(MethodImplOptions.AggressiveInlining)]
87         get
88         {
89             var root = GetTreeRoot();
90             if (GreaterOrEqualThan(index, GetSize(root)))
91             {
92                 return Zero;
93             }
94             while (!EqualToZero(root))
95             {
96                 var left = GetLeftOrDefault(root);
97                 var leftSize = GetSizeOrZero(left);
98                 if (LessThan(index, leftSize))
99                 {
100                     root = left;
101                     continue;
102                 }
103                 if (AreEqual(index, leftSize))
104                 {
105                     return root;
106                 }
107                 root = GetRightOrDefault(root);
108                 index = Subtract(index, Increment(leftSize));
109             }
110             return Zero; // TODO: Impossible situation exception (only if tree structure
111             ↳ broken)
112         }
113     }
114
115     /// <summary>
116     /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
117     ↳ (концом).
118     /// </summary>
119     /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
120     /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
121     /// <returns>Индекс искомой связи.</returns>
122     [MethodImpl(MethodImplOptions.AggressiveInlining)]
123     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)) //
123             ↪ node.Key < root.Key
124         {
125             root = GetLeftOrDefault(root);
126         }
127         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
128             ↪ node.Key > root.Key
129         {
130             root = GetRightOrDefault(root);
131         }
132         else // node.Key == root.Key
133         {
134             return root;
135         }
136     }
137     return Zero;
138 }
139
140 // TODO: Return indices range instead of references count
141 [MethodImpl(MethodImplOptions.AggressiveInlining)]
142 public TLink CountUsages(TLink link)
143 {
144     var root = GetTreeRoot();
145     var total = GetSize(root);
146     var totalRightIgnore = Zero;
147     while (!EqualToZero(root))
148     {
149         var @base = GetBasePartValue(root);
150         if (LessOrEqualThan(@base, link))
151         {
152             root = GetRightOrDefault(root);
153         }
154         else
155         {
156             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
157             root = GetLeftOrDefault(root);
158         }
159     }
160     root = GetTreeRoot();
161     var totalLeftIgnore = Zero;
162     while (!EqualToZero(root))
163     {
164         var @base = GetBasePartValue(root);
165         if (GreaterOrEqualThan(@base, link))
166         {
167             root = GetLeftOrDefault(root);
168         }
169         else
170         {
171             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
172             root = GetRightOrDefault(root);
173         }
174     }
175     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
176 }
177
178 [MethodImpl(MethodImplOptions.AggressiveInlining)]
179 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
180     ↪ EachUsageCore(@base, GetTreeRoot(), handler);
181
182 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
183     ↪ low-level MSIL stack.
184 [MethodImpl(MethodImplOptions.AggressiveInlining)]
185 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
186 {
187     var @continue = Continue;
188     if (EqualToZero(link))
189     {
190         return @continue;
191     }
192     var linkBasePart = GetBasePartValue(link);
193     var @break = Break;

```

```

190     if (GreaterThan(linkBasePart, @base))
191     {
192         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
193         {
194             return @break;
195         }
196     }
197     else if (LessThan(linkBasePart, @base))
198     {
199         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
200         {
201             return @break;
202         }
203     }
204     else //if (linkBasePart == @base)
205     {
206         if (AreEqual(handler(GetLinkValues(link)), @break))
207         {
208             return @break;
209         }
210         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
211         {
212             return @break;
213         }
214         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
215         {
216             return @break;
217         }
218     }
219     return @continue;
220 }
221
222 [MethodImpl(MethodImplOptions.AggressiveInlining)]
223 protected override void PrintNodeValue(TLink node, StringBuilder sb)
224 {
225     ref var link = ref GetLinkDataPartReference(node);
226     sb.Append(' ');
227     sb.Append(link.Source);
228     sb.Append('-');
229     sb.Append('>');
230     sb.Append(link.Target);
231 }
232 }
233 }

```

1.30 ./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 }

```

```

32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 protected abstract TLink GetTreeRoot();
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 protected abstract TLink GetBasePartValue(TLink link);
37
38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
40     ↪ rootSource, TLink rootTarget);
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
44     ↪ rootSource, TLink rootTarget);
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
48     ↪ AsRef<LinksHeader<TLink>>(Header);
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
52     ↪ AsRef<RawLinkDataPart<TLink>>(LinksDataParts + (RawLinkDataPart<TLink>.SizeInBytes *
53     ↪ _addressToInt64Converter.Convert(link)));
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
57     ↪ ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
58     ↪ (RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link)));
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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
69 {
70     ref var firstLink = ref GetLinkDataPartReference(first);
71     ref var secondLink = ref GetLinkDataPartReference(second);
72     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
73     ↪ secondLink.Source, secondLink.Target);
74 }
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
78 {
79     ref var firstLink = ref GetLinkDataPartReference(first);
80     ref var secondLink = ref GetLinkDataPartReference(second);
81     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
82     ↪ secondLink.Source, secondLink.Target);
83 }
84
85 public TLink this[TLink index]
86 {
87     [MethodImpl(MethodImplOptions.AggressiveInlining)]
88     get
89     {
90         var root = GetTreeRoot();
91         if (GreaterOrEqualThan(index, GetSize(root)))
92         {
93             return Zero;
94         }
95         while (!EqualToZero(root))
96         {
97             var left = GetLeftOrDefault(root);
98             var leftSize = GetSizeOrZero(left);
99             if (LessThan(index, leftSize))
100             {
101                 root = left;
102                 continue;
103             }
104             if (AreEqual(index, leftSize))
105             {
106                 return root;
107             }
108             root = GetRightOrDefault(root);
109         }
110     }
111 }

```

```

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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
176 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
177     ↳ EachUsageCore(@base, GetTreeRoot(), handler);
178
179 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
180     ↳ low-level MSIL stack.
181 [MethodImpl(MethodImplOptions.AggressiveInlining)]
182 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
183 {
184     var @continue = Continue;
185     if (EqualToZero(link))
186     {
187         return @continue;
188     }
189     var linkBasePart = GetBasePartValue(link);
190     var @break = Break;
191     if (GreaterThan(linkBasePart, @base))
192     {
193         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
194         {
195             return @break;
196         }
197     }
198     else if (LessThan(linkBasePart, @base))
199     {
200         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
201         {
202             return @break;
203         }
204     }
205     else //if (linkBasePart == @base)
206     {
207         if (AreEqual(handler(GetLinkValues(link)), @break))
208         {
209             return @break;
210         }
211         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
212         {
213             return @break;
214         }
215         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
216         {
217             return @break;
218         }
219     }
220     return @continue;
221 }
222
223 [MethodImpl(MethodImplOptions.AggressiveInlining)]
224 protected override void PrintNodeValue(TLink node, StringBuilder sb)
225 {
226     ref var link = ref GetLinkDataPartReference(node);
227     sb.Append(' ');
228     sb.Append(link.Source);
229     sb.Append('-');
230     sb.Append('>');
231     sb.Append(link.Target);
232 }
233 }

```

1.31 ./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)]

```

```

13     protected override ref TLink GetLeftReference(TLink node) => ref
14         ↳ GetLinkIndexPartReference(node).LeftAsSource;
15
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     protected override ref TLink GetRightReference(TLink node) => ref
18         ↳ GetLinkIndexPartReference(node).RightAsSource;
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     protected override TLink GetLeft(TLink node) =>
22         ↳ GetLinkIndexPartReference(node).LeftAsSource;
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     protected override TLink GetRight(TLink node) =>
26         ↳ GetLinkIndexPartReference(node).RightAsSource;
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     protected override void SetLeft(TLink node, TLink left) =>
30         ↳ GetLinkIndexPartReference(node).LeftAsSource = left;
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     protected override void SetRight(TLink node, TLink right) =>
34         ↳ GetLinkIndexPartReference(node).RightAsSource = right;
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected override TLink GetSize(TLink node) =>
38         ↳ GetLinkIndexPartReference(node).SizeAsSource;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override void SetSize(TLink node, TLink size) =>
42         ↳ GetLinkIndexPartReference(node).SizeAsSource = size;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override TLink GetBasePartValue(TLink link) =>
49         ↳ GetLinkDataPartReference(link).Source;
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
53         ↳ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
54         ↳ (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
58         ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
59         ↳ (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     protected override void ClearNode(TLink node)
63     {
64         ref var link = ref GetLinkIndexPartReference(node);
65         link.LeftAsSource = Zero;
66         link.RightAsSource = Zero;
67         link.SizeAsSource = Zero;
68     }
69 }

```

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

```

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

```



```

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

```

1.33 ./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> :
    ↪ ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
    ↪ constants, byte* linksDataParts, byte* linksIndexParts, byte* header) :
    ↪ base(constants, linksDataParts, linksIndexParts, header) { }
11
12        [MethodImpl(MethodImplOptions.AggressiveInlining)]
13        protected override ref TLink GetLeftReference(TLink node) => ref
    ↪ GetLinkIndexPartReference(node).LeftAsTarget;
14
15        [MethodImpl(MethodImplOptions.AggressiveInlining)]
16        protected override ref TLink GetRightReference(TLink node) => ref
    ↪ GetLinkIndexPartReference(node).RightAsTarget;

```

```

17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     protected override TLink GetLeft(TLink node) =>
19         ↪ GetLinkIndexPartReference(node).LeftAsTarget;
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     protected override TLink GetRight(TLink node) =>
23         ↪ GetLinkIndexPartReference(node).RightAsTarget;
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     protected override void SetLeft(TLink node, TLink left) =>
27         ↪ GetLinkIndexPartReference(node).LeftAsTarget = left;
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     protected override void SetRight(TLink node, TLink right) =>
31         ↪ GetLinkIndexPartReference(node).RightAsTarget = right;
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override TLink GetSize(TLink node) =>
35         ↪ GetLinkIndexPartReference(node).SizeAsTarget;
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override void SetSize(TLink node, TLink size) =>
39         ↪ GetLinkIndexPartReference(node).SizeAsTarget = size;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override TLink GetBasePartValue(TLink link) =>
46         ↪ GetLinkDataPartReference(link).Target;
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
50         ↪ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
51         ↪ (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) ||
56         ↪ (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     protected override void ClearNode(TLink node)
60     {
61         ref var link = ref GetLinkIndexPartReference(node);
62         link.LeftAsTarget = Zero;
63         link.RightAsTarget = Zero;
64         link.SizeAsTarget = Zero;
65     }
66 }

```

1.34 ./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)]

```

```

19     protected override TLink GetLeft(TLink node) =>
20         ↪ GetLinkIndexPartReference(node).LeftAsTarget;
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected override TLink GetRight(TLink node) =>
24         ↪ GetLinkIndexPartReference(node).RightAsTarget;
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected override void SetLeft(TLink node, TLink left) =>
28         ↪ GetLinkIndexPartReference(node).LeftAsTarget = left;
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     protected override void SetRight(TLink node, TLink right) =>
32         ↪ GetLinkIndexPartReference(node).RightAsTarget = right;
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected override TLink GetSize(TLink node) =>
36         ↪ GetLinkIndexPartReference(node).SizeAsTarget;
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override void SetSize(TLink node, TLink size) =>
40         ↪ GetLinkIndexPartReference(node).SizeAsTarget = size;
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override TLink GetBasePartValue(TLink link) =>
47         ↪ GetLinkDataPartReference(link).Target;
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override 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 GetLinkIndexPartReference(node);
63         link.LeftAsTarget = Zero;
64         link.RightAsTarget = Zero;
65         link.SizeAsTarget = Zero;
66     }
67 }

```

1.35 ./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)

```

```

25 {
26     LinksDataParts = linksDataParts;
27     LinksIndexParts = linksIndexParts;
28     Header = header;
29     Break = constants.Break;
30     Continue = constants.Continue;
31 }
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 protected abstract TLink GetTreeRoot(TLink link);
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 protected abstract TLink GetBasePartValue(TLink link);
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 protected abstract TLink GetKeyPartValue(TLink link);
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
    ↪ AsRef<RawLinkDataPart<TLink>>(LinksDataParts + (RawLinkDataPart<TLink>.SizeInBytes *
    ↪ _addressToInt64Converter.Convert(link)));
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
    ↪ ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
    ↪ (RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link)));
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
    ↪ LessThan(GetKeyPartValue(first), GetKeyPartValue(second));
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
    ↪ GreaterThan(GetKeyPartValue(first), GetKeyPartValue(second));
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
56 {
57     ref var link = ref GetLinkDataPartReference(linkIndex);
58     return new Link<TLink>(linkIndex, link.Source, link.Target);
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
            ↪ broken)
88     }
89 }
90
91 /// <summary>
92 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
    ↪ (концом).
93 /// </summary>
94 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
95 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>

```

```

96     /// <returns>Индекс искомой связи.</returns>
97     [MethodImpl(MethodImplOptions.AggressiveInlining)]
98     public abstract TLink Search(TLink source, TLink target);
99
100    [MethodImpl(MethodImplOptions.AggressiveInlining)]
101    protected TLink SearchCore(TLink root, TLink key)
102    {
103        while (!EqualToZero(root))
104        {
105            var rootKey = GetKeyPartValue(root);
106            if (LessThan(key, rootKey)) // node.Key < root.Key
107            {
108                root = GetLeftOrDefault(root);
109            }
110            else if (GreaterThan(key, rootKey)) // node.Key > root.Key
111            {
112                root = GetRightOrDefault(root);
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) =>
128        ↪ EachUsageCore(@base, GetTreeRoot(@base), handler);
129
130    // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
131    ↪ low-level MSIL stack.
132    [MethodImpl(MethodImplOptions.AggressiveInlining)]
133    private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
134    {
135        var @continue = Continue;
136        if (EqualToZero(link))
137        {
138            return @continue;
139        }
140        var @break = Break;
141        if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
142        {
143            return @break;
144        }
145        if (AreEqual(handler(GetLinkValues(link)), @break))
146        {
147            return @break;
148        }
149        if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
150        {
151            return @break;
152        }
153        return @continue;
154    }
155
156    [MethodImpl(MethodImplOptions.AggressiveInlining)]
157    protected override void PrintNodeValue(TLink node, StringBuilder sb)
158    {
159        ref var link = ref GetLinkDataPartReference(node);
160        sb.Append(' ');
161        sb.Append(link.Source);
162        sb.Append('-');
163        sb.Append('>');
164        sb.Append(link.Target);
165    }
166 }

```

1.36 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSizeBalancedTreeMethodsBase.cs

```

1 using System;
2 using System.Text;
3 using System.Collections.Generic;
4 using System.Runtime.CompilerServices;
5 using Platform.Collections.Methods.Trees;

```

```

6  using Platform.Converters;
7  using static System.Runtime.CompilerServices.Unsafe;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Memory.Split.Generic
12 {
13     public unsafe abstract class InternalLinksSizeBalancedTreeMethodsBase<TLink> :
14         ↳ SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
15     {
16         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
17             ↳ UncheckedConverter<TLink, long>.Default;
18
19         protected readonly TLink Break;
20         protected readonly TLink Continue;
21         protected readonly byte* LinksDataParts;
22         protected readonly byte* LinksIndexParts;
23         protected readonly byte* Header;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected InternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants,
27             ↳ byte* linksDataParts, byte* linksIndexParts, byte* header)
28         {
29             LinksDataParts = linksDataParts;
30             LinksIndexParts = linksIndexParts;
31             Header = header;
32             Break = constants.Break;
33             Continue = constants.Continue;
34         }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected abstract TLink GetTreeRoot(TLink link);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected abstract TLink GetBasePartValue(TLink link);
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected abstract TLink GetKeyPartValue(TLink link);
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
47             ↳ AsRef<RawLinkDataPart<TLink>>(LinksDataParts + (RawLinkDataPart<TLink>.SizeInBytes *
48                 ↳ _addressToInt64Converter.Convert(link)));
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
52             ↳ ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
53                 ↳ (RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link)));
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
57             ↳ LessThan(GetKeyPartValue(first), GetKeyPartValue(second));
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
61             ↳ GreaterThan(GetKeyPartValue(first), GetKeyPartValue(second));
62
63         [MethodImpl(MethodImplOptions.AggressiveInlining)]
64         protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
65         {
66             ref var link = ref GetLinkDataPartReference(linkIndex);
67             return new Link<TLink>(linkIndex, link.Source, link.Target);
68         }
69
70         public TLink this[TLink link, TLink index]
71         {
72             [MethodImpl(MethodImplOptions.AggressiveInlining)]
73             get
74             {
75                 var root = GetTreeRoot(link);
76                 if (GreaterOrEqualThan(index, GetSize(root)))
77                 {
78                     return Zero;
79                 }
80                 while (!EqualToZero(root))
81                 {
82                     var left = GetLeftOrDefault(root);
83                     var leftSize = GetSizeOrZero(left);
84                     if (LessThan(index, leftSize))
85                     {

```

```

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
    ↳ broken)
88 }
89 }
90
91 /// <summary>
92 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
    ↳ (концом).
93 /// </summary>
94 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
95 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
96 /// <returns>Индекс искомой связи.</returns>
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 public abstract TLink Search(TLink source, TLink target);
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 protected TLink SearchCore(TLink root, TLink key)
102 {
103     while (!EqualToZero(root))
104     {
105         var rootKey = GetKeyPartValue(root);
106         if (LessThan(key, rootKey)) // node.Key < root.Key
107         {
108             root = GetLeftOrDefault(root);
109         }
110         else if (GreaterThan(key, rootKey)) // node.Key > root.Key
111         {
112             root = GetRightOrDefault(root);
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 }

```

1.37 ./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> :
13         ↪ RelativeCircularDoublyLinkedListMethods<TLink>
14     {
15         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
16             ↪ UncheckedConverter<TLink, long>.Default;
17         private readonly byte* _linksDataParts;
18         private readonly byte* _linksIndexParts;
19         protected readonly TLink Break;
20         protected readonly TLink Continue;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public InternalLinksSourcesLinkedListMethods(LinksConstants<TLink> constants, byte*
24             ↪ linksDataParts, byte* linksIndexParts)
25         {
26             _linksDataParts = linksDataParts;
27             _linksIndexParts = linksIndexParts;
28             Break = constants.Break;
29             Continue = constants.Continue;
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
34             ↪ AsRef<RawLinkDataPart<TLink>>(_linksDataParts + (RawLinkDataPart<TLink>.SizeInBytes
35             ↪ * _addressToInt64Converter.Convert(link)));
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
39             ↪ ref AsRef<RawLinkIndexPart<TLink>>(_linksIndexParts +
40             ↪ (RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link)));
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override TLink GetFirst(TLink head) =>
44             ↪ GetLinkIndexPartReference(head).RootAsSource;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override TLink GetLast(TLink head)
48         {
49             var first = GetLinkIndexPartReference(head).RootAsSource;
50             if (EqualToZero(first))
51             {
52                 return first;
53             }
54             else
55             {
56                 return GetPrevious(first);
57             }
58         }
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         protected override TLink GetPrevious(TLink element) =>
62             ↪ GetLinkIndexPartReference(element).LeftAsSource;
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

56     protected override TLink GetNext(TLink element) =>
57         ↪ GetLinkIndexPartReference(element).RightAsSource;
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override TLink GetSize(TLink head) =>
61         ↪ GetLinkIndexPartReference(head).SizeAsSource;
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override void SetFirst(TLink head, TLink element) =>
65         ↪ GetLinkIndexPartReference(head).RootAsSource = element;
66
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     protected override void SetLast(TLink head, TLink element)
69     {
70         //var first = GetLinkIndexPartReference(head).RootAsSource;
71         //if (EqualToZero(first))
72         //{
73             //    SetFirst(head, element);
74         //}
75         //else
76         //{
77             //    SetPrevious(first, element);
78         //}
79     }
80
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     protected override void SetPrevious(TLink element, TLink previous) =>
83         ↪ GetLinkIndexPartReference(element).LeftAsSource = previous;
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     protected override void SetNext(TLink element, TLink next) =>
87         ↪ GetLinkIndexPartReference(element).RightAsSource = next;
88
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     protected override void SetSize(TLink head, TLink size) =>
91         ↪ GetLinkIndexPartReference(head).SizeAsSource = size;
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     public TLink CountUsages(TLink head) => GetSize(head);
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
98     {
99         ref var link = ref GetLinkDataPartReference(linkIndex);
100         return new Link<TLink>(linkIndex, link.Source, link.Target);
101     }
102
103     [MethodImpl(MethodImplOptions.AggressiveInlining)]
104     public TLink EachUsage(TLink source, Func<IList<TLink>, TLink> handler)
105     {
106         var @continue = Continue;
107         var @break = Break;
108         var current = GetFirst(source);
109         var first = current;
110         while (!EqualToZero(current))
111         {
112             if (AreEqual(handler(GetLinkValues(current)), @break))
113             {
114                 return @break;
115             }
116             current = GetNext(current);
117             if (AreEqual(current, first))
118             {
119                 return @continue;
120             }
121         }
122         return @continue;
123     }
124 }

```

1.38 ./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> :
  ↳ InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
8 {
9     [MethodImpl(MethodImplOptions.AggressiveInlining)]
10    public InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
  ↳ constants, byte* linksDataParts, byte* linksIndexParts, byte* header) :
  ↳ base(constants, linksDataParts, linksIndexParts, header) { }
11
12    [MethodImpl(MethodImplOptions.AggressiveInlining)]
13    protected override ref TLink GetLeftReference(TLink node) => ref
  ↳ GetLinkIndexPartReference(node).LeftAsSource;
14
15    [MethodImpl(MethodImplOptions.AggressiveInlining)]
16    protected override ref TLink GetRightReference(TLink node) => ref
  ↳ GetLinkIndexPartReference(node).RightAsSource;
17
18    [MethodImpl(MethodImplOptions.AggressiveInlining)]
19    protected override TLink GetLeft(TLink node) =>
  ↳ GetLinkIndexPartReference(node).LeftAsSource;
20
21    [MethodImpl(MethodImplOptions.AggressiveInlining)]
22    protected override TLink GetRight(TLink node) =>
  ↳ GetLinkIndexPartReference(node).RightAsSource;
23
24    [MethodImpl(MethodImplOptions.AggressiveInlining)]
25    protected override void SetLeft(TLink node, TLink left) =>
  ↳ GetLinkIndexPartReference(node).LeftAsSource = left;
26
27    [MethodImpl(MethodImplOptions.AggressiveInlining)]
28    protected override void SetRight(TLink node, TLink right) =>
  ↳ GetLinkIndexPartReference(node).RightAsSource = right;
29
30    [MethodImpl(MethodImplOptions.AggressiveInlining)]
31    protected override TLink GetSize(TLink node) =>
  ↳ GetLinkIndexPartReference(node).SizeAsSource;
32
33    [MethodImpl(MethodImplOptions.AggressiveInlining)]
34    protected override void SetSize(TLink node, TLink size) =>
  ↳ GetLinkIndexPartReference(node).SizeAsSource = size;
35
36    [MethodImpl(MethodImplOptions.AggressiveInlining)]
37    protected override TLink GetTreeRoot(TLink link) =>
  ↳ GetLinkIndexPartReference(link).RootAsSource;
38
39    [MethodImpl(MethodImplOptions.AggressiveInlining)]
40    protected override TLink GetBasePartValue(TLink link) =>
  ↳ GetLinkDataPartReference(link).Source;
41
42    [MethodImpl(MethodImplOptions.AggressiveInlining)]
43    protected override TLink GetKeyPartValue(TLink link) =>
  ↳ GetLinkDataPartReference(link).Target;
44
45    [MethodImpl(MethodImplOptions.AggressiveInlining)]
46    protected override void ClearNode(TLink node)
47    {
48        ref var link = ref GetLinkIndexPartReference(node);
49        link.LeftAsSource = Zero;
50        link.RightAsSource = Zero;
51        link.SizeAsSource = Zero;
52    }
53
54    public override TLink Search(TLink source, TLink target) =>
  ↳ SearchCore(GetTreeRoot(source), target);
55 }
56 }

```

1.39 ./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> :
  ↳ InternalLinksSizeBalancedTreeMethodsBase<TLink>
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

10     public InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants,
11         ↪ byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
12         ↪ linksDataParts, linksIndexParts, header) { }
13
14     [MethodImpl(MethodImplOptions.AggressiveInlining)]
15     protected override ref TLink GetLeftReference(TLink node) => ref
16         ↪ GetLinkIndexPartReference(node).LeftAsSource;
17
18     [MethodImpl(MethodImplOptions.AggressiveInlining)]
19     protected override ref TLink GetRightReference(TLink node) => ref
20         ↪ GetLinkIndexPartReference(node).RightAsSource;
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected override TLink GetLeft(TLink node) =>
24         ↪ GetLinkIndexPartReference(node).LeftAsSource;
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected override TLink GetRight(TLink node) =>
28         ↪ GetLinkIndexPartReference(node).RightAsSource;
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     protected override void SetLeft(TLink node, TLink left) =>
32         ↪ GetLinkIndexPartReference(node).LeftAsSource = left;
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected override void SetRight(TLink node, TLink right) =>
36         ↪ GetLinkIndexPartReference(node).RightAsSource = right;
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override TLink GetSize(TLink node) =>
40         ↪ GetLinkIndexPartReference(node).SizeAsSource;
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override void SetSize(TLink node, TLink size) =>
44         ↪ GetLinkIndexPartReference(node).SizeAsSource = size;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override TLink GetTreeRoot(TLink link) =>
48         ↪ GetLinkIndexPartReference(link).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 TLink GetKeyPartValue(TLink link) =>
56         ↪ GetLinkDataPartReference(link).Target;
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     protected override void ClearNode(TLink node)
60     {
61         ref var link = ref GetLinkIndexPartReference(node);
62         link.LeftAsSource = Zero;
63         link.RightAsSource = Zero;
64         link.SizeAsSource = Zero;
65     }
66
67     public override TLink Search(TLink source, TLink target) =>
68         ↪ SearchCore(GetTreeRoot(source), target);
69 }
70 }

```

1.40 ./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)]

```

```

13     protected override ref TLink GetLeftReference(TLink node) => ref
14         ↳ GetLinkIndexPartReference(node).LeftAsTarget;
15
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     protected override ref TLink GetRightReference(TLink node) => ref
18         ↳ GetLinkIndexPartReference(node).RightAsTarget;
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     protected override TLink GetLeft(TLink node) =>
22         ↳ GetLinkIndexPartReference(node).LeftAsTarget;
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     protected override TLink GetRight(TLink node) =>
26         ↳ GetLinkIndexPartReference(node).RightAsTarget;
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     protected override void SetLeft(TLink node, TLink left) =>
30         ↳ GetLinkIndexPartReference(node).LeftAsTarget = left;
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     protected override void SetRight(TLink node, TLink right) =>
34         ↳ GetLinkIndexPartReference(node).RightAsTarget = right;
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected override TLink GetSize(TLink node) =>
38         ↳ GetLinkIndexPartReference(node).SizeAsTarget;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override void SetSize(TLink node, TLink size) =>
42         ↳ GetLinkIndexPartReference(node).SizeAsTarget = size;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override TLink GetTreeRoot(TLink link) =>
46         ↳ GetLinkIndexPartReference(link).RootAsTarget;
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected override TLink GetBasePartValue(TLink link) =>
50         ↳ GetLinkDataPartReference(link).Target;
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected override TLink GetKeyPartValue(TLink link) =>
54         ↳ GetLinkDataPartReference(link).Source;
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected override void ClearNode(TLink node)
58     {
59         ref var link = ref GetLinkIndexPartReference(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 }

```

1.41 ./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)]

```

```

16     protected override ref TLink GetRightReference(TLink node) => ref
17         ↳ GetLinkIndexPartReference(node).RightAsTarget;
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     protected override TLink GetLeft(TLink node) =>
21         ↳ GetLinkIndexPartReference(node).LeftAsTarget;
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     protected override TLink GetRight(TLink node) =>
25         ↳ GetLinkIndexPartReference(node).RightAsTarget;
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     protected override void SetLeft(TLink node, TLink left) =>
29         ↳ GetLinkIndexPartReference(node).LeftAsTarget = left;
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override void SetRight(TLink node, TLink right) =>
33         ↳ GetLinkIndexPartReference(node).RightAsTarget = right;
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override TLink GetSize(TLink node) =>
37         ↳ GetLinkIndexPartReference(node).SizeAsTarget;
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected override void SetSize(TLink node, TLink size) =>
41         ↳ GetLinkIndexPartReference(node).SizeAsTarget = size;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override TLink GetTreeRoot(TLink link) =>
45         ↳ GetLinkIndexPartReference(link).RootAsTarget;
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override TLink GetBasePartValue(TLink link) =>
49         ↳ GetLinkDataPartReference(link).Target;
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override TLink GetKeyPartValue(TLink link) =>
53         ↳ GetLinkDataPartReference(link).Source;
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override void ClearNode(TLink node)
57     {
58         ref var link = ref GetLinkIndexPartReference(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 }

```

1.42 ./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     }

```

```

24 [MethodImpl(MethodImplOptions.AggressiveInlining)]
25 public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }
26
27 [MethodImpl(MethodImplOptions.AggressiveInlining)]
28 public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
    ↳ memoryReservationStep, Default<LinksConstants<TLink>>.Instance,
    ↳ IndexTreeType.Default, useLinkedList: true) { }
29
30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
    ↳ this(dataMemory, indexMemory, memoryReservationStep, constants,
    ↳ IndexTreeType.Default, useLinkedList: true) { }
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants,
    ↳ IndexTreeType indexTreeType, bool useLinkedList) : base(dataMemory, indexMemory,
    ↳ memoryReservationStep, constants, useLinkedList)
35 {
36     if (indexTreeType == IndexTreeType.SizeBalancedTree)
37     {
38         _createInternalSourceTreeMethods = () => new
            ↳ InternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
39         _createExternalSourceTreeMethods = () => new
            ↳ ExternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
40         _createInternalTargetTreeMethods = () => new
            ↳ InternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
41         _createExternalTargetTreeMethods = () => new
            ↳ ExternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
42     }
43     else
44     {
45         _createInternalSourceTreeMethods = () => new
            ↳ InternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
46         _createExternalSourceTreeMethods = () => new
            ↳ ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
47         _createInternalTargetTreeMethods = () => new
            ↳ InternalLinksTargetsRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
48         _createExternalTargetTreeMethods = () => new
            ↳ ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods<TLink>(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
49     }
50     Init(dataMemory, indexMemory);
51 }
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected override void SetPointers(IResizableDirectMemory dataMemory,
    ↳ IResizableDirectMemory indexMemory)
55 {
56     _linksDataParts = (byte*)dataMemory.Pointer;
57     _linksIndexParts = (byte*)indexMemory.Pointer;
58     _header = _linksIndexParts;
59     if (_useLinkedList)
60     {
61         InternalSourcesListMethods = new
            ↳ InternalLinksSourcesLinkedListMethods<TLink>(Constants, _linksDataParts,
            ↳ _linksIndexParts);
62     }
63     else
64     {
65         InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
66     }
67     ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
68     InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
69     ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
70     UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_linksDataParts, _header);
71 }

```

```

72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     protected override void ResetPointers()
74     {
75         base.ResetPointers();
76         _linksDataParts = null;
77         _linksIndexParts = null;
78         _header = null;
79     }
80
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     protected override ref LinksHeader<TLink> GetHeaderReference() => ref
83     ↪ AsRef<LinksHeader<TLink>>(_header);
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
87     ↪ => ref AsRef<RawLinkDataPart<TLink>>(_linksDataParts + (LinkDataPartSizeInBytes *
88     ↪ ConvertToInt64(linkIndex)));
89
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
92     ↪ linkIndex) => ref AsRef<RawLinkIndexPart<TLink>>(_linksIndexParts +
93     ↪ (LinkIndexPartSizeInBytes * ConvertToInt64(linkIndex)));
94 }
95 }

```

1.43 ./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
54         // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
55         ↪ нужно использовать не список а дерево, так как так можно быстрее проверить на
56         ↪ наличие связи внутри

```

```

50     protected ILinksListMethods<TLink> UnusedLinksListMethods;
51
52     /// <summary>
53     /// Возвращает общее число связей находящихся в хранилище.
54     /// </summary>
55     protected virtual TLink Total
56     {
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         get
59         {
60             ref var header = ref GetHeaderReference();
61             return Subtract(header.AllocatedLinks, header.FreeLinks);
62         }
63     }
64
65     public virtual LinksConstants<TLink> Constants
66     {
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         get;
69     }
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
73     ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants, bool
74     ↪ useLinkedList)
75     {
76         _dataMemory = dataMemory;
77         _indexMemory = indexMemory;
78         _dataMemoryReservationStepInBytes = memoryReservationStep * LinkDataPartSizeInBytes;
79         _indexMemoryReservationStepInBytes = memoryReservationStep *
80         ↪ LinkIndexPartSizeInBytes;
81         _useLinkedList = useLinkedList;
82         Constants = constants;
83     }
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
87     ↪ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
88     ↪ memoryReservationStep, Default<LinksConstants<TLink>>.Instance, useLinkedList: true)
89     ↪ { }
90
91     [MethodImpl(MethodImplOptions.AggressiveInlining)]
92     protected virtual void Init(IResizableDirectMemory dataMemory, IResizableDirectMemory
93     ↪ indexMemory)
94     {
95         // Read allocated links from header
96         if (indexMemory.ReservedCapacity < LinkHeaderSizeInBytes)
97         {
98             indexMemory.ReservedCapacity = LinkHeaderSizeInBytes;
99         }
100         SetPointers(dataMemory, indexMemory);
101         ref var header = ref GetHeaderReference();
102         var allocatedLinks = ConvertToInt64(header.AllocatedLinks);
103         // Adjust reserved capacity
104         var minimumDataReservedCapacity = allocatedLinks * LinkDataPartSizeInBytes;
105         if (minimumDataReservedCapacity < dataMemory.UsedCapacity)
106         {
107             minimumDataReservedCapacity = dataMemory.UsedCapacity;
108         }
109         if (minimumDataReservedCapacity < _dataMemoryReservationStepInBytes)
110         {
111             minimumDataReservedCapacity = _dataMemoryReservationStepInBytes;
112         }
113         var minimumIndexReservedCapacity = allocatedLinks * LinkDataPartSizeInBytes;
114         if (minimumIndexReservedCapacity < indexMemory.UsedCapacity)
115         {
116             minimumIndexReservedCapacity = indexMemory.UsedCapacity;
117         }
118         if (minimumIndexReservedCapacity < _indexMemoryReservationStepInBytes)
119         {
120             minimumIndexReservedCapacity = _indexMemoryReservationStepInBytes;
121         }
122         // Check for alignment
123         if (minimumDataReservedCapacity % _dataMemoryReservationStepInBytes > 0)
124         {
125             minimumDataReservedCapacity = ((minimumDataReservedCapacity /
126             ↪ _dataMemoryReservationStepInBytes) * _dataMemoryReservationStepInBytes) +
127             ↪ _dataMemoryReservationStepInBytes;
128         }
129     }

```



```

120     if (minimumIndexReservedCapacity % _indexMemoryReservationStepInBytes > 0)
121     {
122         minimumIndexReservedCapacity = ((minimumIndexReservedCapacity /
123             ↪ _indexMemoryReservationStepInBytes) * _indexMemoryReservationStepInBytes) +
124             ↪ _indexMemoryReservationStepInBytes;
125     }
126     if (dataMemory.ReservedCapacity != minimumDataReservedCapacity)
127     {
128         dataMemory.ReservedCapacity = minimumDataReservedCapacity;
129     }
130     if (indexMemory.ReservedCapacity != minimumIndexReservedCapacity)
131     {
132         indexMemory.ReservedCapacity = minimumIndexReservedCapacity;
133     }
134     SetPointers(dataMemory, indexMemory);
135     header = ref GetHeaderReference();
136     // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
137     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
138     dataMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
139         ↪ LinkDataPartSizeInBytes) + LinkDataPartSizeInBytes; // First link is read only
140     ↪ zero link.
141     indexMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
142         ↪ LinkIndexPartSizeInBytes) + LinkHeaderSizeInBytes;
143     // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
144     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
145     header.ReservedLinks = ConvertToAddress((dataMemory.ReservedCapacity -
146         ↪ LinkDataPartSizeInBytes) / LinkDataPartSizeInBytes);
147 }
148
149 [MethodImpl(MethodImplOptions.AggressiveInlining)]
150 public virtual TLink Count(IList<TLink> restrictions)
151 {
152     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
153     if (restrictions.Count == 0)
154     {
155         return Total;
156     }
157     var constants = Constants;
158     var any = constants.Any;
159     var index = restrictions[constants.IndexPart];
160     if (restrictions.Count == 1)
161     {
162         if (AreEqual(index, any))
163         {
164             return Total;
165         }
166         return Exists(index) ? GetOne() : GetZero();
167     }
168     if (restrictions.Count == 2)
169     {
170         var value = restrictions[1];
171         if (AreEqual(index, any))
172         {
173             if (AreEqual(value, any))
174             {
175                 return Total; // Any - как отсутствие ограничения
176             }
177             var externalReferencesRange = constants.ExternalReferencesRange;
178             if (externalReferencesRange.HasValue &&
179                 ↪ externalReferencesRange.Value.Contains(value))
180             {
181                 return Add(ExternalSourcesTreeMethods.CountUsages(value),
182                     ↪ ExternalTargetsTreeMethods.CountUsages(value));
183             }
184             else
185             {
186                 if (_useLinkedList)
187                 {
188                     return Add(InternalSourcesListMethods.CountUsages(value),
189                         ↪ InternalTargetsTreeMethods.CountUsages(value));
190                 }
191                 else
192                 {
193                     return Add(InternalSourcesTreeMethods.CountUsages(value),
194                         ↪ InternalTargetsTreeMethods.CountUsages(value));
195                 }
196             }
197         }
198     }
199 }

```

```

188     else
189     {
190         if (!Exists(index))
191         {
192             return GetZero();
193         }
194         if (AreEqual(value, any))
195         {
196             return GetOne();
197         }
198         ref var storedLinkValue = ref GetLinkDataPartReference(index);
199         if (AreEqual(storedLinkValue.Source, value) ||
200             ↪ AreEqual(storedLinkValue.Target, value))
201         {
202             return GetOne();
203         }
204         return GetZero();
205     }
206 if (restrictions.Count == 3)
207 {
208     var externalReferencesRange = constants.ExternalReferencesRange;
209     var source = restrictions[constants.SourcePart];
210     var target = restrictions[constants.TargetPart];
211     if (AreEqual(index, any))
212     {
213         if (AreEqual(source, any) && AreEqual(target, any))
214         {
215             return Total;
216         }
217         else if (AreEqual(source, any))
218         {
219             if (externalReferencesRange.HasValue &&
220                 ↪ externalReferencesRange.Value.Contains(target))
221             {
222                 return ExternalTargetsTreeMethods.CountUsages(target);
223             }
224             else
225             {
226                 return InternalTargetsTreeMethods.CountUsages(target);
227             }
228         }
229         else if (AreEqual(target, any))
230         {
231             if (externalReferencesRange.HasValue &&
232                 ↪ externalReferencesRange.Value.Contains(source))
233             {
234                 return ExternalSourcesTreeMethods.CountUsages(source);
235             }
236             else
237             {
238                 if (_useLinkedList)
239                 {
240                     return InternalSourcesListMethods.CountUsages(source);
241                 }
242                 else
243                 {
244                     return InternalSourcesTreeMethods.CountUsages(source);
245                 }
246             }
247         }
248     }
249     else //if(source != Any && target != Any)
250     {
251         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
252         TLink link;
253         if (externalReferencesRange.HasValue)
254         {
255             if (externalReferencesRange.Value.Contains(source) &&
256                 ↪ externalReferencesRange.Value.Contains(target))
257             {
258                 link = ExternalSourcesTreeMethods.Search(source, target);
259             }
260             else if (externalReferencesRange.Value.Contains(source))
261             {
262                 link = InternalTargetsTreeMethods.Search(source, target);
263             }
264             else if (externalReferencesRange.Value.Contains(target))
265             {
266                 link = InternalSourcesTreeMethods.Search(source, target);
267             }
268         }
269         else
270         {
271             link = InternalSourcesListMethods.Search(source, target);
272         }
273     }
274 }

```

```

262         if (_useLinkedList)
263         {
264             link = ExternalSourcesTreeMethods.Search(source, target);
265         }
266         else
267         {
268             link = InternalSourcesTreeMethods.Search(source, target);
269         }
270     }
271     else
272     {
273         if (_useLinkedList ||
274             ↪ GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
275             ↪ InternalTargetsTreeMethods.CountUsages(target)))
276         {
277             link = InternalTargetsTreeMethods.Search(source, target);
278         }
279         else
280         {
281             link = InternalSourcesTreeMethods.Search(source, target);
282         }
283     }
284 }
285 else
286 {
287     if (_useLinkedList ||
288         ↪ GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
289         ↪ InternalTargetsTreeMethods.CountUsages(target)))
290     {
291         link = InternalTargetsTreeMethods.Search(source, target);
292     }
293     else
294     {
295         link = InternalSourcesTreeMethods.Search(source, target);
296     }
297 }
298 return AreEqual(link, constants.Null) ? GetZero() : GetOne();
299 }
300 }
301 else
302 {
303     if (!Exists(index))
304     {
305         return GetZero();
306     }
307     if (AreEqual(source, any) && AreEqual(target, any))
308     {
309         return GetOne();
310     }
311     ref var storedLinkValue = ref GetLinkDataPartReference(index);
312     if (!AreEqual(source, any) && !AreEqual(target, any))
313     {
314         if (AreEqual(storedLinkValue.Source, source) &&
315             ↪ AreEqual(storedLinkValue.Target, target))
316         {
317             return GetOne();
318         }
319         return GetZero();
320     }
321     var value = default(TLink);
322     if (AreEqual(source, any))
323     {
324         value = target;
325     }
326     if (AreEqual(target, any))
327     {
328         value = source;
329     }
330     if (AreEqual(storedLinkValue.Source, value) ||
331         ↪ AreEqual(storedLinkValue.Target, value))
332     {
333         return GetOne();
334     }
335     return GetZero();
336 }
337 }
338 throw new NotSupportedException("Другие размеры и способы ограничений не
339     ↪ поддерживаются.");

```

```

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             return @break;
350         }
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)
368     {
369         var value = restrictions[1];
370         if (AreEqual(index, any))
371         {
372             if (AreEqual(value, any))
373             {
374                 return Each(handler, Array.Empty<TLink>());
375             }
376             if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
377             {
378                 return @break;
379             }
380             return Each(handler, new Link<TLink>(index, any, value));
381         }
382         else
383         {
384             if (!Exists(index))
385             {
386                 return @continue;
387             }
388             if (AreEqual(value, any))
389             {
390                 return handler(GetLinkStruct(index));
391             }
392             ref var storedLinkValue = ref GetLinkDataPartReference(index);
393             if (AreEqual(storedLinkValue.Source, value) ||
394                 AreEqual(storedLinkValue.Target, value))
395             {
396                 return handler(GetLinkStruct(index));
397             }
398             return @continue;
399         }
400     }
401     if (restrictions.Count == 3)
402     {
403         var externalReferencesRange = constants.ExternalReferencesRange;
404         var source = restrictions[constants.SourcePart];
405         var target = restrictions[constants.TargetPart];
406         if (AreEqual(index, any))
407         {
408             if (AreEqual(source, any) && AreEqual(target, any))
409             {
410                 return Each(handler, Array.Empty<TLink>());
411             }

```

```

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;
445         if (externalReferencesRange.HasValue)
446         {
447             if (externalReferencesRange.Value.Contains(source) &&
448                 ↪ externalReferencesRange.Value.Contains(target))
449             {
450                 link = ExternalSourcesTreeMethods.Search(source, target);
451             }
452             else if (externalReferencesRange.Value.Contains(source))
453             {
454                 link = InternalTargetsTreeMethods.Search(source, target);
455             }
456             else if (externalReferencesRange.Value.Contains(target))
457             {
458                 if (_useLinkedList)
459                 {
460                     link = ExternalSourcesTreeMethods.Search(source, target);
461                 }
462                 else
463                 {
464                     link = InternalSourcesTreeMethods.Search(source, target);
465                 }
466             }
467             else
468             {
469                 if (_useLinkedList ||
470                     ↪ GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
471                     ↪ InternalTargetsTreeMethods.CountUsages(target)))
472                 {
473                     link = InternalTargetsTreeMethods.Search(source, target);
474                 }
475                 else
476                 {
477                     link = InternalSourcesTreeMethods.Search(source, target);
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     }

```

```

482         else
483         {
484             link = InternalSourcesTreeMethods.Search(source, target);
485         }
486     }
487     return AreEqual(link, constants.Null) ? @continue :
488         ↪ handler(GetLinkStruct(link));
489 }
490 else
491 {
492     if (!Exists(index))
493     {
494         return @continue;
495     }
496     if (AreEqual(source, any) && AreEqual(target, any))
497     {
498         return handler(GetLinkStruct(index));
499     }
500     ref var storedLinkValue = ref GetLinkDataPartReference(index);
501     if (!AreEqual(source, any) && !AreEqual(target, any))
502     {
503         if (AreEqual(storedLinkValue.Source, source) &&
504             AreEqual(storedLinkValue.Target, target))
505         {
506             return handler(GetLinkStruct(index));
507         }
508         return @continue;
509     }
510     var value = default(TLink);
511     if (AreEqual(source, any))
512     {
513         value = target;
514     }
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
528     ↪ поддерживаются.");
529 }
530 /// <remarks>
531 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
532     ↪ в другом месте (но не в менеджере памяти, а в логике Links)
533 /// </remarks>
534 [MethodImpl(MethodImplOptions.AggressiveInlining)]
535 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
536 {
537     var constants = Constants;
538     var @null = constants.Null;
539     var externalReferencesRange = constants.ExternalReferencesRange;
540     var linkIndex = restrictions[constants.IndexPart];
541     ref var link = ref GetLinkDataPartReference(linkIndex);
542     var source = link.Source;
543     var target = link.Target;
544     ref var header = ref GetHeaderReference();
545     ref var rootAsSource = ref header.RootAsSource;
546     ref var rootAsTarget = ref header.RootAsTarget;
547     // Будет корректно работать только в том случае, если пространство выделенной связи
548     ↪ предварительно заполнено нулями
549     if (!AreEqual(source, @null))
550     {
551         if (externalReferencesRange.HasValue &&
552             ↪ externalReferencesRange.Value.Contains(source))
553         {
554             ExternalSourcesTreeMethods.Detach(ref rootAsSource, linkIndex);
555         }
556     }
557     else
558     {
559         if (_useLinkedList)
560         {
561             return handler(GetLinkStruct(linkIndex));
562         }
563     }
564 }

```

```

556         {
557             InternalSourcesListMethods.Detach(source, linkIndex);
558         }
559         else
560         {
561             InternalSourcesTreeMethods.Detach(ref
562                 ↪ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
563         }
564     }
565     if (!AreEqual(target, @null))
566     {
567         if (externalReferencesRange.HasValue &&
568             ↪ externalReferencesRange.Value.Contains(target))
569         {
570             ExternalTargetsTreeMethods.Detach(ref rootAsTarget, linkIndex);
571         }
572         else
573         {
574             InternalTargetsTreeMethods.Detach(ref
575                 ↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
576         }
577     }
578     source = link.Source = substitution[constants.SourcePart];
579     target = link.Target = substitution[constants.TargetPart];
580     if (!AreEqual(source, @null))
581     {
582         if (externalReferencesRange.HasValue &&
583             ↪ externalReferencesRange.Value.Contains(source))
584         {
585             ExternalSourcesTreeMethods.Attach(ref rootAsSource, linkIndex);
586         }
587         else
588         {
589             if (_useLinkedList)
590             {
591                 InternalSourcesListMethods.AttachAsLast(source, linkIndex);
592             }
593             else
594             {
595                 InternalSourcesTreeMethods.Attach(ref
596                     ↪ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
597             }
598         }
599     }
600     if (!AreEqual(target, @null))
601     {
602         if (externalReferencesRange.HasValue &&
603             ↪ externalReferencesRange.Value.Contains(target))
604         {
605             ExternalTargetsTreeMethods.Attach(ref rootAsTarget, linkIndex);
606         }
607         else
608         {
609             InternalTargetsTreeMethods.Attach(ref
610                 ↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
611         }
612     }
613     return linkIndex;
614 }
615
616 /// <remarks>
617 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
618 ↪ пространство
619 /// </remarks>
620 [MethodImpl(MethodImplOptions.AggressiveInlining)]
621 public virtual TLink Create(ICollection<TLink> restrictions)
622 {
623     ref var header = ref GetHeaderReference();
624     var freeLink = header.FirstFreeLink;
625     if (!AreEqual(freeLink, Constants.Null))
626     {
627         UnusedLinksListMethods.Detach(freeLink);
628     }
629     else
630     {
631         var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
632         if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))

```

```

626     {
627         throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
628     }
629     if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
630     {
631         _dataMemory.ReservedCapacity += _dataMemoryReservationStepInBytes;
632         _indexMemory.ReservedCapacity += _indexMemoryReservationStepInBytes;
633         SetPointers(_dataMemory, _indexMemory);
634         header = ref GetHeaderReference();
635         header.ReservedLinks = ConvertToAddress(_dataMemory.ReservedCapacity /
        ↪ LinkDataPartSizeInBytes);
636     }
637     freeLink = header.AllocatedLinks = Increment(header.AllocatedLinks);
638     _dataMemory.UsedCapacity += LinkDataPartSizeInBytes;
639     _indexMemory.UsedCapacity += LinkIndexPartSizeInBytes;
640 }
641 return freeLink;
642 }
643
644 [MethodImpl(MethodImplOptions.AggressiveInlining)]
645 public virtual void Delete(IList<TLink> restrictions)
646 {
647     ref var header = ref GetHeaderReference();
648     var link = restrictions[Constants.IndexPart];
649     if (LessThan(link, header.AllocatedLinks))
650     {
651         UnusedLinksListMethods.AttachAsFirst(link);
652     }
653     else if (AreEqual(link, header.AllocatedLinks))
654     {
655         header.AllocatedLinks = Decrement(header.AllocatedLinks);
656         _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
657         _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
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
672 [MethodImpl(MethodImplOptions.AggressiveInlining)]
673 public IList<TLink> GetLinkStruct(TLink linkIndex)
674 {
675     ref var link = ref GetLinkDataPartReference(linkIndex);
676     return new Link<TLink>(linkIndex, link.Source, link.Target);
677 }
678
679 /// <remarks>
680 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
681 ↪ адрес реально поменялся
682 ///
683 /// Указатель this.links может быть в том же месте,
684 /// так как 0-я связь не используется и имеет такой же размер как Header,
685 /// поэтому header размещается в том же месте, что и 0-я связь
686 /// </remarks>
687 [MethodImpl(MethodImplOptions.AggressiveInlining)]
688 protected abstract void SetPointers(IResizableDirectMemory dataMemory,
689     ↪ IResizableDirectMemory indexMemory);
690
691 [MethodImpl(MethodImplOptions.AggressiveInlining)]
692 protected virtual void ResetPointers()
693 {
694     InternalSourcesListMethods = null;
695     InternalSourcesTreeMethods = null;
696     ExternalSourcesTreeMethods = null;
697     InternalTargetsTreeMethods = null;
698     ExternalTargetsTreeMethods = null;
699     UnusedLinksListMethods = null;
700 }
701
702 [MethodImpl(MethodImplOptions.AggressiveInlining)]
703 protected abstract ref LinksHeader<TLink> GetHeaderReference();

```



```

700 [MethodImpl(MethodImplOptions.AggressiveInlining)]
701 protected abstract ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex);
702
703 [MethodImpl(MethodImplOptions.AggressiveInlining)]
704 protected abstract ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
705     ↪ linkIndex);
706
707 [MethodImpl(MethodImplOptions.AggressiveInlining)]
708 protected virtual bool Exists(TLink link)
709     => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
710     && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
711     && !IsUnusedLink(link);
712
713 [MethodImpl(MethodImplOptions.AggressiveInlining)]
714 protected virtual bool IsUnusedLink(TLink linkIndex)
715 {
716     if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
717         ↪ is not needed
718     {
719         // TODO: Reduce access to memory in different location (should be enough to use
720         ↪ just linkIndexPart)
721         ref var linkDataPart = ref GetLinkDataPartReference(linkIndex);
722         ref var linkIndexPart = ref GetLinkIndexPartReference(linkIndex);
723         return AreEqual(linkIndexPart.SizeAsTarget, default) &&
724             ↪ !AreEqual(linkDataPart.Source, default);
725     }
726     else
727     {
728         return true;
729     }
730 }
731
732 [MethodImpl(MethodImplOptions.AggressiveInlining)]
733 protected virtual TLink GetOne() => _one;
734
735 [MethodImpl(MethodImplOptions.AggressiveInlining)]
736 protected virtual TLink GetZero() => default;
737
738 [MethodImpl(MethodImplOptions.AggressiveInlining)]
739 protected virtual bool AreEqual(TLink first, TLink second) =>
740     ↪ _equalityComparer.Equals(first, second);
741
742 [MethodImpl(MethodImplOptions.AggressiveInlining)]
743 protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
744     ↪ second) < 0;
745
746 [MethodImpl(MethodImplOptions.AggressiveInlining)]
747 protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
748     ↪ _comparer.Compare(first, second) <= 0;
749
750 [MethodImpl(MethodImplOptions.AggressiveInlining)]
751 protected virtual bool GreaterThan(TLink first, TLink second) =>
752     ↪ _comparer.Compare(first, second) > 0;
753
754 [MethodImpl(MethodImplOptions.AggressiveInlining)]
755 protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
756     ↪ _comparer.Compare(first, second) >= 0;
757
758 [MethodImpl(MethodImplOptions.AggressiveInlining)]
759 protected virtual long ConvertToInt64(TLink value) =>
760     ↪ _addressToInt64Converter.Convert(value);
761
762 [MethodImpl(MethodImplOptions.AggressiveInlining)]
763 protected virtual TLink ConvertToAddress(long value) =>
764     ↪ _int64ToAddressConverter.Convert(value);
765
766 [MethodImpl(MethodImplOptions.AggressiveInlining)]
767 protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
768     ↪ second);
769
770 [MethodImpl(MethodImplOptions.AggressiveInlining)]
771 protected virtual TLink Subtract(TLink first, TLink second) =>
772     ↪ Arithmetic<TLink>.Subtract(first, second);
773
774 [MethodImpl(MethodImplOptions.AggressiveInlining)]
775 protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);

```

```

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 }

```

1.44 ./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)]

```

```

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

```

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

```

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

```

1.46 ./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 }

```

1.47 ./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 :
        ↳ ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
10     {
11         protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
12         protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
13         protected new readonly LinksHeader<TLink>* Header;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected
        ↳ UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
        ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
        ↳ linksIndexParts, LinksHeader<TLink>* header)
        ↳ : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
17         {
18             LinksDataParts = linksDataParts;
19             LinksIndexParts = linksIndexParts;
20             Header = header;
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override TLink GetZero() => 0U;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override bool EqualToZero(TLink value) => value == 0U;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override bool AreEqual(TLink first, TLink second) => first == second;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override bool GreaterThanZero(TLink value) => value > 0U;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override bool GreaterThan(TLink first, TLink second) => first > second;
37
38

```

```

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

```

1.48 ./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;

```

```

14 [MethodImpl(MethodImplOptions.AggressiveInlining)]
15 protected UInt32 ExternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink>
16     ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
    ↪ 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 TLink GetZero() => 0U;
26
27 [MethodImpl(MethodImplOptions.AggressiveInlining)]
28 protected override bool EqualToZero(TLink value) => value == 0U;
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 > 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
    ↪ always true for ulong
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
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     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
87     {
88         ref var firstLink = ref LinksDataParts[first];
89         ref var secondLink = ref LinksDataParts[second];
90         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
91             ↪ secondLink.Source, secondLink.Target);
92     }
93 }
94 }

```

1.49 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods

```

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 :
9          ↪ UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public
13             ↪ UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(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 FirstIsToTheLeftOfSecond(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)

```

```

53     {
54         ref var link = ref LinksIndexParts[node];
55         link.LeftAsSource = Zero;
56         link.RightAsSource = Zero;
57         link.SizeAsSource = Zero;
58     }
59 }
60 }

```

1.50 ./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)]
26         protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetLeft(TLink node, TLink left) =>
33             ↳ LinksIndexParts[node].LeftAsSource = left;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override void SetRight(TLink node, TLink right) =>
37             ↳ LinksIndexParts[node].RightAsSource = right;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override void SetSize(TLink node, TLink size) =>
44             ↳ LinksIndexParts[node].SizeAsSource = size;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override TLink GetTreeRoot() => Header->RootAsSource;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
54             ↳ TLink secondSource, TLink secondTarget)
55             => firstSource < secondSource || firstSource == secondSource && firstTarget <
56             ↳ secondTarget;
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
60             ↳ TLink secondSource, TLink secondTarget)
61             => firstSource > secondSource || firstSource == secondSource && firstTarget >
62             ↳ secondTarget;
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected override void ClearNode(TLink node)
66         {
67             ref var link = ref LinksIndexParts[node];
68             link.LeftAsSource = Zero;
69             link.RightAsSource = Zero;
70         }
71     }
72 }

```



```

57         link.SizeAsSource = Zero;
58     }
59 }
60 }

```

1.51 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods

```

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)]
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() => Header->RootAsTarget;
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected override bool FirstIsToLeftOfSecond(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 bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
61             ↳ TLink secondSource, TLink secondTarget)
62             => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
63             ↳ secondSource;
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         protected override void ClearNode(TLink node)
67         {
68             ref var link = ref LinksIndexParts[node];
69             link.LeftAsTarget = Zero;
70             link.RightAsTarget = Zero;
71             link.SizeAsTarget = Zero;
72         }
73     }
74 }

```

```
60 }
```

1.52 ./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 }
```

1.53 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase.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 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     [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 RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
71     ↳ ref LinksDataParts[link];
```

```

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

```

1.54 ./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             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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool LessOrEqualThanZero(TLink value) => value == 0UL; // value is
51             ↪ always >= 0 for ulong
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override bool LessThanZero(TLink value) => false; // value < 0 is always false
58             ↪ for ulong
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         protected override bool LessThan(TLink first, TLink second) => first < second;
62
63         [MethodImpl(MethodImplOptions.AggressiveInlining)]
64         protected override TLink Increment(TLink value) => ++value;
65     }

```

```

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) =>
71         ↪ ref LinksDataParts[link];
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
75         ↪ ref LinksIndexParts[link];
76
77     [MethodImpl(MethodImplOptions.AggressiveInlining)]
78     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
79         ↪ GetKeyPartValue(first) < GetKeyPartValue(second);
80
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
83         ↪ GetKeyPartValue(first) > GetKeyPartValue(second);
84 }
85 }

```

1.55 ./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>
10     {
11         private readonly RawLinkDataPart<TLink>* _linksDataParts;
12         private readonly RawLinkIndexPart<TLink>* _linksIndexParts;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public UInt32InternalLinksSourcesLinkedListMethods(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
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
25             ↪ ref _linksDataParts[link];
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
29             ↪ ref _linksIndexParts[link];
30     }
31 }

```

1.56 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods.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.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)]

```

```

14     protected override ref TLink GetLeftReference(TLink node) => ref
15         ↳ LinksIndexParts[node].LeftAsSource;
16
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     protected override ref TLink GetRightReference(TLink node) => ref
19         ↳ LinksIndexParts[node].RightAsSource;
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     protected override void SetLeft(TLink node, TLink left) =>
29         ↳ LinksIndexParts[node].LeftAsSource = left;
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override void SetRight(TLink node, TLink right) =>
33         ↳ LinksIndexParts[node].RightAsSource = right;
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override void SetSize(TLink node, TLink size) =>
40         ↳ LinksIndexParts[node].SizeAsSource = size;
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override TLink GetTreeRoot(TLink node) => LinksIndexParts[node].RootAsSource;
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Target;
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     public override TLink Search(TLink source, TLink target) =>
61         ↳ SearchCore(GetTreeRoot(source), target);
62 }

```

1.57 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesSizeBalancedTreeMethod

```

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

```

```

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(TLink node) => LinksIndexParts[node].RootAsSource;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Target;
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override void ClearNode(TLink node)
51     {
52         ref var link = ref LinksIndexParts[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 }

```

1.58 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksTargetsRecursionlessSizeBalance

```

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     }

```

```

30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected override void SetSize(TLink node, TLink size) =>
36         ↳ LinksIndexParts[node].SizeAsTarget = size;
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override TLink GetTreeRoot(TLink node) => LinksIndexParts[node].RootAsTarget;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Source;
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override void ClearNode(TLink node)
49     {
50         ref var link = ref LinksIndexParts[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 }

```

1.59 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksTargetsSizeBalancedTreeMethod

```

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

```



```

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

```

1.60 ./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
24             indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public UInt32SplitMemoryLinks(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 UInt32SplitMemoryLinks(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 UInt32SplitMemoryLinks(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                     UInt32InternalLinksSourcesSizeBalancedTreeMethods(Constants,
48                     indexMemory, _linksDataParts, _linksIndexParts, _header);
49                 _createExternalSourceTreeMethods = () => new
50                     UInt32ExternalLinksSourcesSizeBalancedTreeMethods(Constants,
51                     indexMemory, _linksDataParts, _linksIndexParts, _header);
52                 _createInternalTargetTreeMethods = () => new
53                     UInt32InternalLinksTargetsSizeBalancedTreeMethods(Constants,
54                     indexMemory, _linksDataParts, _linksIndexParts, _header);
55                 _createExternalTargetTreeMethods = () => new
56                     UInt32ExternalLinksTargetsSizeBalancedTreeMethods(Constants,
57                     indexMemory, _linksDataParts, _linksIndexParts, _header);
58             }
59         }
60     }
61 }

```

```

40     }
41     else
42     {
43         _createInternalSourceTreeMethods = () => new
44             ↳ UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
45             ↳ _linksDataParts, _linksIndexParts, _header);
46         _createExternalSourceTreeMethods = () => new
47             ↳ UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
48             ↳ _linksDataParts, _linksIndexParts, _header);
49         _createInternalTargetTreeMethods = () => new
50             ↳ UInt32InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
51             ↳ _linksDataParts, _linksIndexParts, _header);
52         _createExternalTargetTreeMethods = () => new
53             ↳ UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
54             ↳ _linksDataParts, _linksIndexParts, _header);
55     }
56     Init(dataMemory, indexMemory);
57 }
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected override void SetPointers(IResizableDirectMemory dataMemory,
61     ↳ IResizableDirectMemory indexMemory)
62 {
63     _linksDataParts = (RawLinkDataPart<TLink>*)dataMemory.Pointer;
64     _linksIndexParts = (RawLinkIndexPart<TLink>*)indexMemory.Pointer;
65     _header = (LinksHeader<TLink>*)indexMemory.Pointer;
66     if (_useLinkedList)
67     {
68         InternalSourcesListMethods = new
69             ↳ UInt32InternalLinksSourcesLinkedListMethods(Constants, _linksDataParts,
70             ↳ _linksIndexParts);
71     }
72     else
73     {
74         InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
75     }
76     ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
77     InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
78     ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
79     UnusedLinksListMethods = new UInt32UnusedLinksListMethods(_linksDataParts, _header);
80 }
81
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 protected override void ResetPointers()
84 {
85     base.ResetPointers();
86     _linksDataParts = null;
87     _linksIndexParts = null;
88     _header = null;
89 }
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
96     ↳ => ref _linksDataParts[linkIndex];
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
100     ↳ linkIndex) => ref _linksIndexParts[linkIndex];
101
102 [MethodImpl(MethodImplOptions.AggressiveInlining)]
103 protected override bool AreEqual(TLink first, TLink second) => first == second;
104
105 [MethodImpl(MethodImplOptions.AggressiveInlining)]
106 protected override bool LessThan(TLink first, TLink second) => first < second;
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
110
111 [MethodImpl(MethodImplOptions.AggressiveInlining)]
112 protected override bool GreaterThan(TLink first, TLink second) => first > second;
113
114 [MethodImpl(MethodImplOptions.AggressiveInlining)]
115 protected override bool GreaterOrEqualThan(TLink first, TLink second) => first >= second;
116
117 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

105     protected override TLink GetZero() => 0U;
106
107     [MethodImpl(MethodImplOptions.AggressiveInlining)]
108     protected override TLink GetOne() => 1U;
109
110     [MethodImpl(MethodImplOptions.AggressiveInlining)]
111     protected override long ConvertToInt64(TLink value) => value;
112
113     [MethodImpl(MethodImplOptions.AggressiveInlining)]
114     protected override TLink ConvertToAddress(long value) => (TLink)value;
115
116     [MethodImpl(MethodImplOptions.AggressiveInlining)]
117     protected override TLink Add(TLink first, TLink second) => first + second;
118
119     [MethodImpl(MethodImplOptions.AggressiveInlining)]
120     protected override TLink Subtract(TLink first, TLink second) => first - second;
121
122     [MethodImpl(MethodImplOptions.AggressiveInlining)]
123     protected override TLink Increment(TLink link) => ++link;
124
125     [MethodImpl(MethodImplOptions.AggressiveInlining)]
126     protected override TLink Decrement(TLink link) => --link;
127 }
128 }

```

1.61 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32UnusedLinksListMethods.cs

```

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

```

1.62 ./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     }
24 }

```

```

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

```

```

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

```

1.63 ./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() => OUL;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override bool EqualToZero(ulong value) => value == OUL;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override bool AreEqual(ulong first, ulong second) => first == second;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override bool GreaterThanZero(ulong value) => value > OUL;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override bool GreaterThan(ulong first, ulong second) => first > second;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
47             ↪ always true for ulong
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool LessOrEqualThanZero(ulong value) => value == OUL; // value is
51             ↪ always >= 0 for ulong
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
58             ↪ for ulong
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         protected override bool LessThan(ulong first, ulong second) => first < second;
62
63         [MethodImpl(MethodImplOptions.AggressiveInlining)]
64         protected override ulong Increment(ulong value) => ++value;
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         protected override ulong Decrement(ulong value) => --value;
68
69         [MethodImpl(MethodImplOptions.AggressiveInlining)]
70         protected override ulong Add(ulong first, ulong second) => first + second;
71
72         [MethodImpl(MethodImplOptions.AggressiveInlining)]
73         protected override ulong Subtract(ulong first, ulong second) => first - second;

```

```

68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 protected override ref LinksHeader<TLink> GetHeaderReference() => ref *Header;
70
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 FirstIsToTheLeftOfSecond(TLink first, TLink second)
81 {
82     ref var firstLink = ref LinksDataParts[first];
83     ref var secondLink = ref LinksDataParts[second];
84     return FirstIsToTheLeftOfSecond(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 }
97 }

```

1.64 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods

```

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     }
48 }

```

```

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 FirstIsToLeftOfSecond(TLink firstSource, TLink firstTarget,
45         ↪ TLink secondSource, TLink secondTarget)
46         => firstSource < secondSource || firstSource == secondSource && firstTarget <
47         ↪ secondTarget;
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
51         ↪ TLink secondSource, TLink secondTarget)
52         => firstSource > secondSource || firstSource == secondSource && firstTarget >
53         ↪ secondTarget;
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 }

```

1.65 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSourcesSizeBalancedTreeMethods

```

1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt64;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      public unsafe class UInt64ExternalLinksSourcesSizeBalancedTreeMethods :
9          ↪ UInt64ExternalLinksSizeBalancedTreeMethodsBase
10      {
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          public UInt64ExternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
13              ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
14              ↪ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
15              ↪ linksIndexParts, header) { }
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override ref TLink GetLeftReference(TLink node) => ref
19              ↪ LinksIndexParts[node].LeftAsSource;
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          protected override ref TLink GetRightReference(TLink node) => ref
23              ↪ LinksIndexParts[node].RightAsSource;
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          protected override void SetLeft(TLink node, TLink left) =>
33              ↪ LinksIndexParts[node].LeftAsSource = left;
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected override void SetRight(TLink node, TLink right) =>
37              ↪ LinksIndexParts[node].RightAsSource = right;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override void SetSize(TLink node, TLink size) =>
44              ↪ LinksIndexParts[node].SizeAsSource = size;
45
46          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47          protected override TLink GetTreeRoot() => Header->RootAsSource;
48
49          [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

41     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
45         ↪ TLink secondSource, TLink secondTarget)
46         => firstSource < secondSource || firstSource == secondSource && firstTarget <
47         ↪ secondTarget;
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
51         ↪ TLink secondSource, TLink secondTarget)
52         => firstSource > secondSource || firstSource == secondSource && firstTarget >
53         ↪ secondTarget;
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 }

```

1.66 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksTargetsRecursionlessSizeBalance

```

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 :
9          ↪ UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public
13         ↪ UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(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() => Header->RootAsTarget;
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

44     protected override bool FirstIsToLeftOfSecond(TLink firstSource, TLink firstTarget,
45         ↪ TLink secondSource, TLink secondTarget)
46         => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
47         ↪ secondSource;
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
51         ↪ TLink secondSource, TLink secondTarget)
52         => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
53         ↪ secondSource;
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 }

```

1.67 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksTargetsSizeBalancedTreeMethods

```

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

```

```

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

```

1.68 ./csharp/Platform.Data.Doublets.Memory.Split.Specific/UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase

```

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 :
         ↪ InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
10     {
11         protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
12         protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
13         protected new readonly LinksHeader<TLink>* Header;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected
         ↪ UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
         ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
         ↪ linksIndexParts, LinksHeader<TLink>* header)
         : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
17         {
18             LinksDataParts = linksDataParts;
19             LinksIndexParts = linksIndexParts;
20             Header = header;
21         }
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
         ↪ always true for ulong
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override bool LessOrEqualThanZero(ulong value) => value == OUL; // value is
         ↪ always >= 0 for ulong
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
         ↪ for ulong
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

55     protected override bool LessThan(ulong first, ulong second) => first < second;
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override ulong Increment(ulong value) => ++value;
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected override ulong Decrement(ulong value) => --value;
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override ulong Add(ulong first, ulong second) => first + second;
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override ulong Subtract(ulong first, ulong second) => first - second;
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
71     ↪ ref LinksDataParts[link];
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
75     ↪ ref LinksIndexParts[link];
76
77     [MethodImpl(MethodImplOptions.AggressiveInlining)]
78     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
79     ↪ GetKeyPartValue(first) < GetKeyPartValue(second);
80
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
83     ↪ GetKeyPartValue(first) > GetKeyPartValue(second);
84 }
85 }

```

1.69 ./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

```

```

44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
46     ↪ always >= 0 for ulong
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
53     ↪ for ulong
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override bool LessThan(ulong first, ulong second) => first < second;
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     protected override ulong Increment(ulong value) => ++value;
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     protected override ulong Decrement(ulong value) => --value;
63
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     protected override ulong Add(ulong first, ulong second) => first + second;
66
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     protected override ulong Subtract(ulong first, ulong second) => first - second;
69
70     [MethodImpl(MethodImplOptions.AggressiveInlining)]
71     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
72     ↪ ref LinksDataParts[link];
73
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
76     ↪ ref LinksIndexParts[link];
77
78     [MethodImpl(MethodImplOptions.AggressiveInlining)]
79     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second) =>
80     ↪ GetKeyPartValue(first) < GetKeyPartValue(second);
81
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
84     ↪ GetKeyPartValue(first) > GetKeyPartValue(second);
85 }
86 }

```

1.70 ./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
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
25         ↪ ref _linksDataParts[link];
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
29         ↪ ref _linksIndexParts[link];
30     }
31 }

```

1.71 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods

```

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
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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override void ClearNode(TLink node)
58         {
59             ref var link = ref LinksIndexParts[node];
60             link.LeftAsSource = Zero;
61             link.RightAsSource = Zero;
62             link.SizeAsSource = Zero;
63         }
64
65         public override TLink Search(TLink source, TLink target) =>
66             ↳ SearchCore(GetTreeRoot(source), target);
67     }
68 }

```

1.72 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesSizeBalancedTreeMethods

```

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

```

```

9 {
10     [MethodImpl(MethodImplOptions.AggressiveInlining)]
11     public UInt64InternalLinksSourcesSizeBalancedTreeMethods(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(TLink node) => LinksIndexParts[node].RootAsSource;
39
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) =>
        ↳ SearchCore(GetTreeRoot(source), target);
56 }
57 }

```

1.73 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods

```

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 :
        ↳ UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public
            ↳ UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(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 ulong GetLeftReference(ulong node) => ref
            ↳ LinksIndexParts[node].LeftAsTarget;

```

```

15     [MethodImpl(MethodImplOptions.AggressiveInlining)]
16     protected override ref ulong GetRightReference(ulong node) => ref
17     ↪ LinksIndexParts[node].RightAsTarget;
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsTarget;
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsTarget;
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     protected override void SetLeft(TLink node, TLink left) =>
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(TLink node) => LinksIndexParts[node].RootAsTarget;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Source;
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override void ClearNode(TLink node)
51     {
52         ref var link = ref LinksIndexParts[node];
53         link.LeftAsTarget = Zero;
54         link.RightAsTarget = Zero;
55         link.SizeAsTarget = Zero;
56     }
57
58     public override TLink Search(TLink source, TLink target) =>
59     ↪ SearchCore(GetTreeRoot(target), source);
60 }

```

1.74 ./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

```

```

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

```

1.75 ./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)]

```



```

32 public UInt64SplitMemoryLinks(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
            ↳ UInt64InternalLinksSourcesSizeBalancedTreeMethods(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
37         _createExternalSourceTreeMethods = () => new
            ↳ UInt64ExternalLinksSourcesSizeBalancedTreeMethods(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
38         _createInternalTargetTreeMethods = () => new
            ↳ UInt64InternalLinksTargetsSizeBalancedTreeMethods(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
39         _createExternalTargetTreeMethods = () => new
            ↳ UInt64ExternalLinksTargetsSizeBalancedTreeMethods(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
40     }
41     else
42     {
43         _createInternalSourceTreeMethods = () => new
            ↳ UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
44         _createExternalSourceTreeMethods = () => new
            ↳ UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
45         _createInternalTargetTreeMethods = () => new
            ↳ UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(Constants,
            ↳ _linksDataParts, _linksIndexParts, _header);
46         _createExternalTargetTreeMethods = () => new
            ↳ UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods(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
            ↳ 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 }

```

1.76 ./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>*
          ↳ 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) =>
          ↳ ref _links[link];
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
27     }
28 }

```

1.77 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksAvlBalancedTreeMethodsBase.cs

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;

```

```

4 using System.Runtime.CompilerServices;
5 using Platform.Collections.Methods.Trees;
6 using Platform.Converters;
7 using Platform.Numbers;
8 using static System.Runtime.CompilerServices.Unsafe;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.Memory.United.Generic
13 {
14     public unsafe abstract class LinksAvlBalancedTreeMethodsBase<TLink> :
15         ↳ SizedAndThreadedAVLBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
16     {
17         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
18             ↳ UncheckedConverter<TLink, long>.Default;
19         private static readonly UncheckedConverter<TLink, int> _addressToInt32Converter =
20             ↳ UncheckedConverter<TLink, int>.Default;
21         private static readonly UncheckedConverter<bool, TLink> _boolToAddressConverter =
22             ↳ UncheckedConverter<bool, TLink>.Default;
23         private static readonly UncheckedConverter<TLink, bool> _addressToBoolConverter =
24             ↳ UncheckedConverter<TLink, bool>.Default;
25         private static readonly UncheckedConverter<int, TLink> _int32ToAddressConverter =
26             ↳ UncheckedConverter<int, TLink>.Default;
27
28         protected readonly TLink Break;
29         protected readonly TLink Continue;
30         protected readonly byte* Links;
31         protected readonly byte* Header;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
35             ↳ byte* header)
36         {
37             Links = links;
38             Header = header;
39             Break = constants.Break;
40             Continue = constants.Continue;
41         }
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected abstract TLink GetTreeRoot();
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected abstract TLink GetBasePartValue(TLink link);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
51             ↳ rootSource, TLink rootTarget);
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
55             ↳ rootSource, TLink rootTarget);
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
59             ↳ AsRef<LinksHeader<TLink>>(Header);
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
63             ↳ AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
64             ↳ _addressToInt64Converter.Convert(link)));
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
68         {
69             ref var link = ref GetLinkReference(linkIndex);
70             return new Link<TLink>(linkIndex, link.Source, link.Target);
71         }
72
73         [MethodImpl(MethodImplOptions.AggressiveInlining)]
74         protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
75         {
76             ref var firstLink = ref GetLinkReference(first);
77             ref var secondLink = ref GetLinkReference(second);
78             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
79             ↳ secondLink.Source, secondLink.Target);
80         }
81
82         [MethodImpl(MethodImplOptions.AggressiveInlining)]
83         protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)

```

```

71 {
72     ref var firstLink = ref GetLinkReference(first);
73     ref var secondLink = ref GetLinkReference(second);
74     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
75 }
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected virtual TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5,
    ↪ -5);
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 protected virtual void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
    ↪ Bit<TLink>.PartialWrite(storedValue, size, 5, -5);
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 protected virtual bool GetLeftIsChildValue(TLink value)
85 {
86     unchecked
87     {
88         return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 4, 1));
89         //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
90     }
91 }
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 protected virtual void SetLeftIsChildValue(ref TLink storedValue, bool value)
95 {
96     unchecked
97     {
98         var previousValue = storedValue;
99         var modified = Bit<TLink>.PartialWrite(previousValue,
    ↪ _boolToAddressConverter.Convert(value), 4, 1);
    storedValue = modified;
100     }
101 }
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 protected virtual bool GetRightIsChildValue(TLink value)
105 {
106     unchecked
107     {
108         return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 3, 1));
109         //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
110     }
111 }
112
113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
114 protected virtual void SetRightIsChildValue(ref TLink storedValue, bool value)
115 {
116     unchecked
117     {
118         var previousValue = storedValue;
119         var modified = Bit<TLink>.PartialWrite(previousValue,
    ↪ _boolToAddressConverter.Convert(value), 3, 1);
    storedValue = modified;
120     }
121 }
122
123 [MethodImpl(MethodImplOptions.AggressiveInlining)]
124 protected bool IsChild(TLink parent, TLink possibleChild)
125 {
126     var parentSize = GetSize(parent);
127     var childSize = GetSizeOrZero(possibleChild);
128     return GreaterThanZero(childSize) && LessOrEqualThan(childSize, parentSize);
129 }
130
131 [MethodImpl(MethodImplOptions.AggressiveInlining)]
132 protected virtual sbyte GetBalanceValue(TLink storedValue)
133 {
134     unchecked
135     {
136         var value = _addressToInt32Converter.Convert(Bit<TLink>.PartialRead(storedValue,
    ↪ 0, 3));
137         value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
    ↪ end of sbyte
138         return (sbyte)value;
139     }
140 }
141

```

```

142 }
143
144 [MethodImpl(MethodImplOptions.AggressiveInlining)]
145 protected virtual void SetBalanceValue(ref TLink storedValue, sbyte value)
146 {
147     unchecked
148     {
149         var packagedValue = _int32ToAddressConverter.Convert((byte)value >> 5 & 4 |
150             ↪ value & 3);
151         var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
152         storedValue = modified;
153     }
154 }
155
156 public TLink this[TLink index]
157 {
158     [MethodImpl(MethodImplOptions.AggressiveInlining)]
159     get
160     {
161         var root = GetTreeRoot();
162         if (GreaterOrEqualThan(index, GetSize(root)))
163         {
164             return Zero;
165         }
166         while (!EqualToZero(root))
167         {
168             var left = GetLeftOrDefault(root);
169             var leftSize = GetSizeOrZero(left);
170             if (LessThan(index, leftSize))
171             {
172                 root = left;
173                 continue;
174             }
175             if (AreEqual(index, leftSize))
176             {
177                 return root;
178             }
179             root = GetRightOrDefault(root);
180             index = Subtract(index, Increment(leftSize));
181         }
182         return Zero; // TODO: Impossible situation exception (only if tree structure
183             ↪ broken)
184     }
185 }
186
187 /// <summary>
188 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
189 ↪ (концом).
190 /// </summary>
191 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
192 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
193 /// <returns>Индекс искомой связи.</returns>
194 [MethodImpl(MethodImplOptions.AggressiveInlining)]
195 public TLink Search(TLink source, TLink target)
196 {
197     var root = GetTreeRoot();
198     while (!EqualToZero(root))
199     {
200         ref var rootLink = ref GetLinkReference(root);
201         var rootSource = rootLink.Source;
202         var rootTarget = rootLink.Target;
203         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
204             ↪ node.Key < root.Key
205         {
206             root = GetLeftOrDefault(root);
207         }
208         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
209             ↪ node.Key > root.Key
210         {
211             root = GetRightOrDefault(root);
212         }
213         else // node.Key == root.Key
214         {
215             return root;
216         }
217     }
218     return Zero;
219 }

```

```

216 // TODO: Return indices range instead of references count
217 [MethodImpl(MethodImplOptions.AggressiveInlining)]
218 public TLink CountUsages(TLink link)
219 {
220     var root = GetTreeRoot();
221     var total = GetSize(root);
222     var totalRightIgnore = Zero;
223     while (!EqualToZero(root))
224     {
225         var @base = GetBasePartValue(root);
226         if (LessOrEqualThan(@base, link))
227         {
228             root = GetRightOrDefault(root);
229         }
230         else
231         {
232             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
233             root = GetLeftOrDefault(root);
234         }
235     }
236     root = GetTreeRoot();
237     var totalLeftIgnore = Zero;
238     while (!EqualToZero(root))
239     {
240         var @base = GetBasePartValue(root);
241         if (GreaterOrEqualThan(@base, link))
242         {
243             root = GetLeftOrDefault(root);
244         }
245         else
246         {
247             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
248             root = GetRightOrDefault(root);
249         }
250     }
251     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
252 }
253
254 [MethodImpl(MethodImplOptions.AggressiveInlining)]
255 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
256 {
257     var root = GetTreeRoot();
258     if (EqualToZero(root))
259     {
260         return Continue;
261     }
262     TLink first = Zero, current = root;
263     while (!EqualToZero(current))
264     {
265         var @base = GetBasePartValue(current);
266         if (GreaterOrEqualThan(@base, link))
267         {
268             if (AreEqual(@base, link))
269             {
270                 first = current;
271             }
272             current = GetLeftOrDefault(current);
273         }
274         else
275         {
276             current = GetRightOrDefault(current);
277         }
278     }
279     if (!EqualToZero(first))
280     {
281         current = first;
282         while (true)
283         {
284             if (AreEqual(handler(GetLinkValues(current)), Break))
285             {
286                 return Break;
287             }
288             current = GetNext(current);
289             if (EqualToZero(current) || !AreEqual(GetBasePartValue(current), link))
290             {
291                 break;
292             }
293         }
294     }

```

```

295     }
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 }

```

1.78 ./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;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected LinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants,
26             ↳ byte* links, byte* header)
27         {
28             Links = links;
29             Header = header;
30             Break = constants.Break;
31             Continue = constants.Continue;
32         }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected abstract TLink GetTreeRoot();
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected abstract TLink GetBasePartValue(TLink link);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
42             ↳ rootSource, TLink rootTarget);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
46             ↳ rootSource, TLink rootTarget);
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
50             ↳ AsRef<LinksHeader<TLink>>(Header);
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
54             ↳ AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
55                 ↳ _addressToInt64Converter.Convert(link)));
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
59         {
60             ref var link = ref GetLinkReference(linkIndex);
61             return new Link<TLink>(linkIndex, link.Source, link.Target);
62         }
63     }
64 }

```

```

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

```



```

127         return root;
128     }
129 }
130 return Zero;
131 }
132
133 // TODO: Return indices range instead of references count
134 [MethodImpl(MethodImplOptions.AggressiveInlining)]
135 public TLink CountUsages(TLink link)
136 {
137     var root = GetTreeRoot();
138     var total = GetSize(root);
139     var totalRightIgnore = Zero;
140     while (!EqualToZero(root))
141     {
142         var @base = GetBasePartValue(root);
143         if (LessOrEqualThan(@base, link))
144         {
145             root = GetRightOrDefault(root);
146         }
147         else
148         {
149             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
150             root = GetLeftOrDefault(root);
151         }
152     }
153     root = GetTreeRoot();
154     var totalLeftIgnore = Zero;
155     while (!EqualToZero(root))
156     {
157         var @base = GetBasePartValue(root);
158         if (GreaterOrEqualThan(@base, link))
159         {
160             root = GetLeftOrDefault(root);
161         }
162         else
163         {
164             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
165             root = GetRightOrDefault(root);
166         }
167     }
168     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
169 }
170
171 [MethodImpl(MethodImplOptions.AggressiveInlining)]
172 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
173     ↳ EachUsageCore(@base, GetTreeRoot(), handler);
174
175 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
176 ↳ low-level MSIL stack.
177 [MethodImpl(MethodImplOptions.AggressiveInlining)]
178 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
179 {
180     var @continue = Continue;
181     if (EqualToZero(link))
182     {
183         return @continue;
184     }
185     var linkBasePart = GetBasePartValue(link);
186     var @break = Break;
187     if (GreaterThan(linkBasePart, @base))
188     {
189         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
190         {
191             return @break;
192         }
193     }
194     else if (LessThan(linkBasePart, @base))
195     {
196         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
197         {
198             return @break;
199         }
200     }
201     else //if (linkBasePart == @base)
202     {
203         if (AreEqual(handler(GetLinkValues(link)), @break))
204         {
205             return @break;
206         }
207     }
208 }

```

```

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 }

```

1.79 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSizeBalancedTreeMethodsBase.cs

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7  using static System.Runtime.CompilerServices.Unsafe;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Memory.United.Generic
12 {
13     public unsafe abstract class LinksSizeBalancedTreeMethodsBase<TLink> :
14         ↳ SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
15     {
16         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
17             ↳ UncheckedConverter<TLink, long>.Default;
18
19         protected readonly TLink Break;
20         protected readonly TLink Continue;
21         protected readonly byte* Links;
22         protected readonly byte* Header;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
26             ↳ byte* header)
27         {
28             Links = links;
29             Header = header;
30             Break = constants.Break;
31             Continue = constants.Continue;
32         }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected abstract TLink GetTreeRoot();
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected abstract TLink GetBasePartValue(TLink link);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
42             ↳ rootSource, TLink rootTarget);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
46             ↳ rootSource, TLink rootTarget);
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
50             ↳ AsRef<LinksHeader<TLink>>(Header);
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

47     protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
48         ↪ AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
49         ↪ _addressToInt64Converter.Convert(link)));
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
53     {
54         ref var link = ref GetLinkReference(linkIndex);
55         return new Link<TLink>(linkIndex, link.Source, link.Target);
56     }
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     protected override bool FirstIsToLeftOfSecond(TLink first, TLink second)
60     {
61         ref var firstLink = ref GetLinkReference(first);
62         ref var secondLink = ref GetLinkReference(second);
63         return FirstIsToLeftOfSecond(firstLink.Source, firstLink.Target,
64         ↪ secondLink.Source, secondLink.Target);
65     }
66
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
69     {
70         ref var firstLink = ref GetLinkReference(first);
71         ref var secondLink = ref GetLinkReference(second);
72         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
73         ↪ secondLink.Source, secondLink.Target);
74     }
75
76     public TLink this[TLink index]
77     {
78         [MethodImpl(MethodImplOptions.AggressiveInlining)]
79         get
80         {
81             var root = GetTreeRoot();
82             if (GreaterOrEqualThan(index, GetSize(root)))
83             {
84                 return Zero;
85             }
86             while (!EqualToZero(root))
87             {
88                 var left = GetLeftOrDefault(root);
89                 var leftSize = GetSizeOrZero(left);
90                 if (LessThan(index, leftSize))
91                 {
92                     root = left;
93                     continue;
94                 }
95                 if (AreEqual(index, leftSize))
96                 {
97                     return root;
98                 }
99                 root = GetRightOrDefault(root);
100                 index = Subtract(index, Increment(leftSize));
101             }
102             return Zero; // TODO: Impossible situation exception (only if tree structure
103             ↪ broken)
104         }
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 GetLinkReference(root);
121             var rootSource = rootLink.Source;
122             var rootTarget = rootLink.Target;
123             if (FirstIsToLeftOfSecond(source, target, rootSource, rootTarget)) //
124             ↪ node.Key < root.Key

```

```

118     {
119         root = GetLeftOrDefault(root);
120     }
121     else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
122         ↪ node.Key > root.Key
123     {
124         root = GetRightOrDefault(root);
125     }
126     else // node.Key == root.Key
127     {
128         return root;
129     }
130     return Zero;
131 }
132
133 // TODO: Return indices range instead of references count
134 [MethodImpl(MethodImplOptions.AggressiveInlining)]
135 public TLink CountUsages(TLink link)
136 {
137     var root = GetTreeRoot();
138     var total = GetSize(root);
139     var totalRightIgnore = Zero;
140     while (!EqualToZero(root))
141     {
142         var @base = GetBasePartValue(root);
143         if (LessOrEqualThan(@base, link))
144         {
145             root = GetRightOrDefault(root);
146         }
147         else
148         {
149             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
150             root = GetLeftOrDefault(root);
151         }
152     }
153     root = GetTreeRoot();
154     var totalLeftIgnore = Zero;
155     while (!EqualToZero(root))
156     {
157         var @base = GetBasePartValue(root);
158         if (GreaterOrEqualThan(@base, link))
159         {
160             root = GetLeftOrDefault(root);
161         }
162         else
163         {
164             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
165             root = GetRightOrDefault(root);
166         }
167     }
168     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
169 }
170
171 [MethodImpl(MethodImplOptions.AggressiveInlining)]
172 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
173     ↪ EachUsageCore(@base, GetTreeRoot(), handler);
174
175 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
176 ↪ low-level MSIL stack.
177 [MethodImpl(MethodImplOptions.AggressiveInlining)]
178 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
179 {
180     var @continue = Continue;
181     if (EqualToZero(link))
182     {
183         return @continue;
184     }
185     var linkBasePart = GetBasePartValue(link);
186     var @break = Break;
187     if (GreaterThan(linkBasePart, @base))
188     {
189         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
190         {
191             return @break;
192         }
193     }
194     else if (LessThan(linkBasePart, @base))
195     {

```

```

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 }

```

1.80 ./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 }

```

```

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

```

1.81 ./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)]

```

```

22     protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     protected override void SetLeft(TLink node, TLink left) =>
26         ↪ GetLinkReference(node).LeftAsSource = left;
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     protected override void SetRight(TLink node, TLink right) =>
30         ↪ GetLinkReference(node).RightAsSource = right;
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsSource;
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override void SetSize(TLink node, TLink size) =>
37         ↪ GetLinkReference(node).SizeAsSource = size;
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
47         ↪ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
48         ↪ (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
52         ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
53         ↪ (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override void ClearNode(TLink node)
57     {
58         ref var link = ref GetLinkReference(node);
59         link.LeftAsSource = Zero;
60         link.RightAsSource = Zero;
61         link.SizeAsSource = Zero;
62     }
63 }

```

1.82 ./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 }

```

```

29
30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsSource;
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 protected override void SetSize(TLink node, TLink size) =>
35     ↳ GetLinkReference(node).SizeAsSource = size;
36
37 [MethodImpl(MethodImplOptions.AggressiveInlining)]
38 protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
39
40 [MethodImpl(MethodImplOptions.AggressiveInlining)]
41 protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
42
43 [MethodImpl(MethodImplOptions.AggressiveInlining)]
44 protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
45     ↳ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
46     ↳ (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
50     ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
51     ↳ (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected override void ClearNode(TLink node)
55 {
56     ref var link = ref GetLinkReference(node);
57     link.LeftAsSource = Zero;
58     link.RightAsSource = Zero;
59     link.SizeAsSource = Zero;
60 }
61 }
62 }

```

1.83 ./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     }
45 }

```



```

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

```

1.84 ./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> :
    ↳ LinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public LinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink> constants,
    ↳ byte* links, byte* header) : base(constants, links, header) { }
11
12        [MethodImpl(MethodImplOptions.AggressiveInlining)]
13        protected override ref TLink GetLeftReference(TLink node) => ref
    ↳ GetLinkReference(node).LeftAsTarget;
14
15        [MethodImpl(MethodImplOptions.AggressiveInlining)]
16        protected override ref TLink GetRightReference(TLink node) => ref
    ↳ GetLinkReference(node).RightAsTarget;
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
20
21        [MethodImpl(MethodImplOptions.AggressiveInlining)]
22        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;

```

```

23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     protected override void SetLeft(TLink node, TLink left) =>
25     ↪ GetLinkReference(node).LeftAsTarget = left;
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     protected override void SetRight(TLink node, TLink right) =>
29     ↪ GetLinkReference(node).RightAsTarget = right;
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsTarget;
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected override void SetSize(TLink node, TLink size) =>
36     ↪ GetLinkReference(node).SizeAsTarget = size;
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
46     ↪ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
47     ↪ (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
51     ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
52     ↪ (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected override void ClearNode(TLink node)
56     {
57         ref var link = ref GetLinkReference(node);
58         link.LeftAsTarget = Zero;
59         link.RightAsTarget = Zero;
60         link.SizeAsTarget = Zero;
61     }
62 }

```

1.85 ./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 }

```

```

30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsTarget;
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 protected override void SetSize(TLink node, TLink size) =>
35     ↳ GetLinkReference(node).SizeAsTarget = size;
36
37 [MethodImpl(MethodImplOptions.AggressiveInlining)]
38 protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;
39
40 [MethodImpl(MethodImplOptions.AggressiveInlining)]
41 protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
42
43 [MethodImpl(MethodImplOptions.AggressiveInlining)]
44 protected override bool FirstIsToLeftOfSecond(TLink firstSource, TLink firstTarget,
45     ↳ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
46     ↳ (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
50     ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
51     ↳ (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected override void ClearNode(TLink node)
55 {
56     ref var link = ref GetLinkReference(node);
57     link.LeftAsTarget = Zero;
58     link.RightAsTarget = Zero;
59     link.SizeAsTarget = Zero;
60 }
61 }
62 }

```

1.86 ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinks.cs

```

1 using System;
2 using System.Runtime.CompilerServices;
3 using Platform.Singletons;
4 using Platform.Memory;
5 using static System.Runtime.CompilerServices.Unsafe;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Memory.United.Generic
10 {
11     public unsafe class UnitedMemoryLinks<TLink> : UnitedMemoryLinksBase<TLink>
12     {
13         private readonly Func<ILinksTreeMethods<TLink>> _createSourceTreeMethods;
14         private readonly Func<ILinksTreeMethods<TLink>> _createTargetTreeMethods;
15         private byte* _header;
16         private byte* _links;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
20
21         /// <summary>
22         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
23         ↳ минимальным шагом расширения базы данных.
24         /// </summary>
25         /// <param name="address">Полный путь к файлу базы данных.</param>
26         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
27         ↳ байтах.</param>
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
30     ↳ FileMappedResizableDirectMemory(address, memoryReservationStep),
31     ↳ memoryReservationStep) { }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
35     ↳ DefaultLinksSizeStep) { }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep) :
39     ↳ this(memory, memoryReservationStep, Default<LinksConstants<TLink>>.Instance,
40     ↳ IndexTreeType.Default) { }
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

1.87 ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinksBase.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Disposables;
5 using Platform.Singletons;
6 using Platform.Converters;
7 using Platform.Numbers;
8 using Platform.Memory;
9 using Platform.Data.Exceptions;
10
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Data.Doublets.Memory.United.Generic
14 {
15     public abstract class UnitedMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16     {
17         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
18         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
19         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            ↳ UncheckedConverter<TLink, long>.Default;
20         private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
            ↳ UncheckedConverter<long, TLink>.Default;
21
22         private static readonly TLink _zero = default;
23         private static readonly TLink _one = Arithmetic.Increment(_zero);
24
25         /// <summary>Возвращает размер одной связи в байтах.</summary>
26         /// <remarks>

```

```

27  /// Используется только во вне класса, не рекомендуется использовать внутри.
28  /// Так как во вне не обязательно будет доступен unsafe C#.
29  /// </remarks>
30  public static readonly long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;
31
32  public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
33
34  public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
35
36  protected readonly IResizableDirectMemory _memory;
37  protected readonly long _memoryReservationStep;
38
39  protected ILinksTreeMethods<TLink> TargetsTreeMethods;
40  protected ILinksTreeMethods<TLink> SourcesTreeMethods;
41  // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
42  //      ↪ нужно использовать не список а дерево, так как так можно быстрее проверить на
43  //      ↪ наличие связи внутри
44  protected ILinksListMethods<TLink> UnusedLinksListMethods;
45
46  /// <summary>
47  /// Возвращает общее число связей находящихся в хранилище.
48  /// </summary>
49  protected virtual TLink Total
50  {
51      [MethodImpl(MethodImplOptions.AggressiveInlining)]
52      get
53      {
54          {
55              ref var header = ref GetHeaderReference();
56              return Subtract(header.AllocatedLinks, header.FreeLinks);
57          }
58      }
59
60  public virtual LinksConstants<TLink> Constants
61  {
62      [MethodImpl(MethodImplOptions.AggressiveInlining)]
63      get;
64  }
65
66  [MethodImpl(MethodImplOptions.AggressiveInlining)]
67  protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
68  ↪ memoryReservationStep, LinksConstants<TLink> constants)
69  {
70      _memory = memory;
71      _memoryReservationStep = memoryReservationStep;
72      Constants = constants;
73  }
74
75  [MethodImpl(MethodImplOptions.AggressiveInlining)]
76  protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
77  ↪ memoryReservationStep) : this(memory, memoryReservationStep,
78  ↪ Default<LinksConstants<TLink>>.Instance) { }
79
80  [MethodImpl(MethodImplOptions.AggressiveInlining)]
81  protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
82  {
83      if (memory.ReservedCapacity < memoryReservationStep)
84      {
85          memory.ReservedCapacity = memoryReservationStep;
86      }
87      SetPointers(memory);
88      ref var header = ref GetHeaderReference();
89      // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
90      memory.UsedCapacity = (Convert.ToInt64(header.AllocatedLinks) * LinkSizeInBytes) +
91      ↪ LinkHeaderSizeInBytes;
92      // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
93      header.ReservedLinks = Convert.ToInt64((memory.ReservedCapacity -
94      ↪ LinkHeaderSizeInBytes) / LinkSizeInBytes);
95  }
96
97  [MethodImpl(MethodImplOptions.AggressiveInlining)]
98  public virtual TLink Count(IList<TLink> restrictions)
99  {
100     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
101     if (restrictions.Count == 0)
102     {
103         return Total;
104     }
105     var constants = Constants;
106     var any = constants.Any;
107     var index = restrictions[constants.IndexPart];

```

```

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

```

```

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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
201 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
202 {
203     var constants = Constants;
204     var @break = constants.Break;
205     if (restrictions.Count == 0)
206     {
207         for (var link = GetOne(); LessOrEqualThan(link,
208             ⇨ GetHeaderReference().AllocatedLinks); link = Increment(link))
209         {
210             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
211             {
212                 return @break;
213             }
214         }
215         return @break;
216     }
217     var @continue = constants.Continue;
218     var any = constants.Any;
219     var index = restrictions[constants.IndexPart];
220     if (restrictions.Count == 1)
221     {
222         if (AreEqual(index, any))
223         {
224             return Each(handler, Array.Empty<TLink>());
225         }
226         if (!Exists(index))
227         {
228             return @continue;
229         }
230         return handler(GetLinkStruct(index));
231     }
232     if (restrictions.Count == 2)
233     {
234         var value = restrictions[1];
235         if (AreEqual(index, any))
236         {
237             if (AreEqual(value, any))
238             {
239                 return Each(handler, Array.Empty<TLink>());
240             }
241             if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
242             {
243                 return @break;
244             }
245             return Each(handler, new Link<TLink>(index, any, value));
246         }
247         else
248         {
249             if (!Exists(index))

```

```

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

```



```

326         throw new NotSupportedException("Другие размеры и способы ограничений не
327         ↳ поддерживаются.");
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))
360         {
361             TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
362         }
363         return linkIndex;
364     }
365     /// <remarks>
366     /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
367     ↳ пространство
368     /// </remarks>
369     [MethodImpl(MethodImplOptions.AggressiveInlining)]
370     public virtual TLink Create(IList<TLink> restrictions)
371     {
372         ref var header = ref GetHeaderReference();
373         var freeLink = header.FirstFreeLink;
374         if (!AreEqual(freeLink, Constants.Null))
375         {
376             UnusedLinksListMethods.Detach(freeLink);
377         }
378         else
379         {
380             var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
381             if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
382             {
383                 throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
384             }
385             if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
386             {
387                 _memory.ReservedCapacity += _memory.ReservationStep;
388                 SetPointers(_memory);
389                 header = ref GetHeaderReference();
390                 header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
391                 ↳ LinkSizeInBytes);
392             }
393             freeLink = header.AllocatedLinks = Increment(header.AllocatedLinks);
394             _memory.UsedCapacity += LinkSizeInBytes;
395         }
396         return freeLink;
397     }
398     [MethodImpl(MethodImplOptions.AggressiveInlining)]
399     public virtual void Delete(IList<TLink> restrictions)
400     {
401         ref var header = ref GetHeaderReference();

```

```

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

```

```

475     protected virtual TLink GetZero() => default;
476
477     [MethodImpl(MethodImplOptions.AggressiveInlining)]
478     protected virtual bool AreEqual(TLink first, TLink second) =>
479         ↪ _equalityComparer.Equals(first, second);
480
481     [MethodImpl(MethodImplOptions.AggressiveInlining)]
482     protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
483         ↪ second) < 0;
484
485     [MethodImpl(MethodImplOptions.AggressiveInlining)]
486     protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
487         ↪ _comparer.Compare(first, second) <= 0;
488
489     [MethodImpl(MethodImplOptions.AggressiveInlining)]
490     protected virtual bool GreaterThan(TLink first, TLink second) =>
491         ↪ _comparer.Compare(first, second) > 0;
492
493     [MethodImpl(MethodImplOptions.AggressiveInlining)]
494     protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
495         ↪ _comparer.Compare(first, second) >= 0;
496
497     [MethodImpl(MethodImplOptions.AggressiveInlining)]
498     protected virtual long ConvertToInt64(TLink value) =>
499         ↪ _addressToInt64Converter.Convert(value);
500
501     [MethodImpl(MethodImplOptions.AggressiveInlining)]
502     protected virtual TLink ConvertToAddress(long value) =>
503         ↪ _int64ToAddressConverter.Convert(value);
504
505     [MethodImpl(MethodImplOptions.AggressiveInlining)]
506     protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
507         ↪ second);
508
509     [MethodImpl(MethodImplOptions.AggressiveInlining)]
510     protected virtual TLink Subtract(TLink first, TLink second) =>
511         ↪ Arithmetic<TLink>.Subtract(first, second);
512
513     [MethodImpl(MethodImplOptions.AggressiveInlining)]
514     protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
515
516     [MethodImpl(MethodImplOptions.AggressiveInlining)]
517     protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
518
519     #region Disposable
520
521     protected override bool AllowMultipleDisposeCalls
522     {
523         [MethodImpl(MethodImplOptions.AggressiveInlining)]
524         get => true;
525     }
526
527     [MethodImpl(MethodImplOptions.AggressiveInlining)]
528     protected override void Dispose(bool manual, bool wasDisposed)
529     {
530         if (!wasDisposed)
531         {
532             ResetPointers();
533             _memory.DisposeIfPossible();
534         }
535     }
536
537     #endregion
538 }
539
540 }

```

1.88 ./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     {

```

```

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

```

1.89 ./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;

```

```

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

```

1.90 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksRecursionlessSizeBalancedTreeMethods

```

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

```

```

43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override bool LessOrEqualThan(uint first, uint second) => first <= second;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override bool LessThanZero(uint value) => false; // value < 0 is always false
48     ↪ for uint
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override bool LessThan(uint first, uint second) => first < second;
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override uint Increment(uint value) => ++value;
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected override uint Decrement(uint value) => --value;
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override uint Add(uint first, uint second) => first + second;
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected override uint Subtract(uint first, uint second) => first - second;
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override bool FirstIsToTheLeftOfSecond(uint first, uint second)
67     {
68         ref var firstLink = ref Links[first];
69         ref var secondLink = ref Links[second];
70         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
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 }

```

1.91 ./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
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override uint GetZero() => 0U;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override bool EqualToZero(uint value) => value == 0U;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override bool AreEqual(uint first, uint second) => first == second;
30     }
31 }

```

```

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

```

1.92 ./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 :
    ↪ UInt32LinksRecursionlessSizeBalancedTreeMethodsBase
8     {
9         public UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<uint>
    ↪ constants, RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links,
    ↪ header) { }
10

```

```

11     [MethodImpl(MethodImplOptions.AggressiveInlining)]
12     protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsSource;
13
14     [MethodImpl(MethodImplOptions.AggressiveInlining)]
15     protected override ref uint GetRightReference(uint node) => ref
16         ↳ Links[node].RightAsSource;
17
18     [MethodImpl(MethodImplOptions.AggressiveInlining)]
19     protected override uint GetLeft(uint node) => Links[node].LeftAsSource;
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     protected override uint GetRight(uint node) => Links[node].RightAsSource;
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     protected override void SetLeft(uint node, uint left) => Links[node].LeftAsSource = left;
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     protected override void SetRight(uint node, uint right) => Links[node].RightAsSource =
29         ↳ right;
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override uint GetSize(uint node) => Links[node].SizeAsSource;
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected override void SetSize(uint node, uint size) => Links[node].SizeAsSource = size;
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override uint GetTreeRoot() => Header->RootAsSource;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override uint GetBasePartValue(uint link) => Links[link].Source;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
45         ↳ uint secondSource, uint secondTarget)
46         => firstSource < secondSource || (firstSource == secondSource && firstTarget <
47             ↳ secondTarget);
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
51         ↳ uint secondSource, uint secondTarget)
52         => firstSource > secondSource || (firstSource == secondSource && firstTarget >
53             ↳ secondTarget);
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override void ClearNode(uint node)
57     {
58         ref var link = ref Links[node];
59         link.LeftAsSource = 0U;
60         link.RightAsSource = 0U;
61         link.SizeAsSource = 0U;
62     }
63 }

```

1.93 ./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)]

```



```

21     protected override uint GetRight(uint node) => Links[node].RightAsSource;
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     protected override void SetLeft(uint node, uint left) => Links[node].LeftAsSource = left;
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected override void SetRight(uint node, uint right) => Links[node].RightAsSource =
28         ↳ right;
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     protected override uint GetSize(uint node) => Links[node].SizeAsSource;
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override void SetSize(uint node, uint size) => Links[node].SizeAsSource = size;
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected override uint GetTreeRoot() => Header->RootAsSource;
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected override uint GetBasePartValue(uint link) => Links[link].Source;
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
44         ↳ uint secondSource, uint secondTarget)
45         => firstSource < secondSource || (firstSource == secondSource && firstTarget <
46         ↳ secondTarget);
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
50         ↳ uint secondSource, uint secondTarget)
51         => firstSource > secondSource || (firstSource == secondSource && firstTarget >
52         ↳ secondTarget);
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected override void ClearNode(uint node)
56     {
57         ref var link = ref Links[node];
58         link.LeftAsSource = 0U;
59         link.RightAsSource = 0U;
60         link.SizeAsSource = 0U;
61     }
62 }

```

1.94 ./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)]

```

```

30     protected override uint GetSize(uint node) => Links[node].SizeAsTarget;
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     protected override void SetSize(uint node, uint size) => Links[node].SizeAsTarget = size;
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override uint GetTreeRoot() => Header->RootAsTarget;
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override uint GetBasePartValue(uint link) => Links[link].Target;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
43         ↪ uint secondSource, uint secondTarget)
44         => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
45             ↪ secondSource);
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override bool FirstIsToTheRightOfSecond(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 void ClearNode(uint node)
55     {
56         ref var link = ref Links[node];
57         link.LeftAsTarget = 0U;
58         link.RightAsTarget = 0U;
59         link.SizeAsTarget = 0U;
60     }
61 }

```

1.95 ./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     }

```

```

41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 protected override bool FirstIsToLeftOfSecond(uint firstSource, uint firstTarget,
43     ↪ uint secondSource, uint secondTarget)
44     => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
45     ↪ secondSource);
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 protected override bool FirstIsToTheRightOfSecond(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 void ClearNode(uint node)
55 {
56     ref var link = ref Links[node];
57     link.LeftAsTarget = 0U;
58     link.RightAsTarget = 0U;
59     link.SizeAsTarget = 0U;
60 }

```

1.96 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32UnitedMemoryLinks.cs

```

1 using System;
2 using System.Runtime.CompilerServices;
3 using Platform.Memory;
4 using Platform.Singletons;
5 using Platform.Data.Doublets.Memory.United.Generic;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Memory.United.Specific
10 {
11     /// <summary>
12     /// <para>Represents a low-level implementation of direct access to resizable memory, for
13     ↪ organizing the storage of links with addresses represented as <see cref="uint" />.</para>
14     /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
15     ↪ размером, для организации хранения связей с адресами представленными в виде <see
16     ↪ cref="uint"/>.</para>
17     /// </summary>
18     public unsafe class UInt32UnitedMemoryLinks : UnitedMemoryLinksBase<uint>
19     {
20         private readonly Func<ILinksTreeMethods<uint>> _createSourceTreeMethods;
21         private readonly Func<ILinksTreeMethods<uint>> _createTargetTreeMethods;
22         private LinksHeader<uint>* _header;
23         private RawLink<uint>* _links;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public UInt32UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
27
28         /// <summary>
29         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
30         ↪ минимальным шагом расширения базы данных.
31         /// </summary>
32         /// <param name="address">Полный путь к файлу базы данных.</param>
33         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
34         ↪ байтах.</param>
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public UInt32UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
37     ↪ FileMappedResizableDirectMemory(address, memoryReservationStep),
38     ↪ memoryReservationStep) { }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public UInt32UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
42     ↪ DefaultLinksSizeStep) { }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
46     ↪ memoryReservationStep) : this(memory, memoryReservationStep,
47     ↪ Default<LinksConstants<uint>>.Instance, IndexTreeType.Default) { }
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
51     ↪ memoryReservationStep, LinksConstants<uint> constants, IndexTreeType indexTreeType)
52     ↪ : base(memory, memoryReservationStep, constants)
53     {
54         if (indexTreeType == IndexTreeType.SizeBalancedTree)

```

```

43     {
44         _createSourceTreeMethods = () => new
45             ↳ UInt32LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
46         _createTargetTreeMethods = () => new
47             ↳ UInt32LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
48     }
49     else
50     {
51         _createSourceTreeMethods = () => new
52             ↳ UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods(Constants, _links,
53             ↳ _header);
54         _createTargetTreeMethods = () => new
55             ↳ UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods(Constants, _links,
56             ↳ _header);
57     }
58     Init(memory, memoryReservationStep);
59 }
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected override void SetPointers(IResizableDirectMemory memory)
63 {
64     _header = (LinksHeader<uint>*)memory.Pointer;
65     _links = (RawLink<uint>*)memory.Pointer;
66     SourcesTreeMethods = _createSourceTreeMethods();
67     TargetsTreeMethods = _createTargetTreeMethods();
68     UnusedLinksListMethods = new UInt32UnusedLinksListMethods(_links, _header);
69 }
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 protected override void ResetPointers()
73 {
74     base.ResetPointers();
75     _links = null;
76     _header = null;
77 }
78
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 protected override ref LinksHeader<uint> GetHeaderReference() => ref *_header;
81
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 protected override ref RawLink<uint> GetLinkReference(uint linkIndex) => ref
84     ↳ _links[linkIndex];
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 protected override bool AreEqual(uint first, uint second) => first == second;
88
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 protected override bool LessThan(uint first, uint second) => first < second;
91
92 [MethodImpl(MethodImplOptions.AggressiveInlining)]
93 protected override bool LessOrEqualThan(uint first, uint second) => first <= second;
94
95 [MethodImpl(MethodImplOptions.AggressiveInlining)]
96 protected override bool GreaterThan(uint first, uint second) => first > second;
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 protected override bool GreaterOrEqualThan(uint first, uint second) => first >= second;
100
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 protected override uint GetZero() => 0U;
103
104 [MethodImpl(MethodImplOptions.AggressiveInlining)]
105 protected override uint GetOne() => 1U;
106
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]
108 protected override long ConvertToInt64(uint value) => (long)value;
109
110 [MethodImpl(MethodImplOptions.AggressiveInlining)]
111 protected override uint ConvertToAddress(long value) => (uint)value;
112
113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
114 protected override uint Add(uint first, uint second) => first + second;
115
116 [MethodImpl(MethodImplOptions.AggressiveInlining)]
117 protected override uint Subtract(uint first, uint second) => first - second;
118
119 [MethodImpl(MethodImplOptions.AggressiveInlining)]
120 protected override uint Increment(uint link) => ++link;

```

```

115     [MethodImpl(MethodImplOptions.AggressiveInlining)]
116     protected override uint Decrement(uint link) => --link;
117 }
118 }

```

1.97 ./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 }

```

1.98 ./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)]

```

```

43     protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
    ↳ always >= 0 for ulong
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
    ↳ for ulong
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override bool LessThan(ulong first, ulong second) => first < second;
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected override ulong Increment(ulong value) => ++value;
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override ulong Decrement(ulong value) => --value;
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected override ulong Add(ulong first, ulong second) => first + second;
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override ulong Subtract(ulong first, ulong second) => first - second;
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
68     {
69         ref var firstLink = ref Links[first];
70         ref var secondLink = ref Links[second];
71         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
    ↳ secondLink.Source, secondLink.Target);
72     }
73
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
76     {
77         ref var firstLink = ref Links[first];
78         ref var secondLink = ref Links[second];
79         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↳ 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 }
```

1.99 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksRecursionlessSizeBalancedTreeMethods

```
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 :
9         ↳ LinksRecursionlessSizeBalancedTreeMethodsBase<ulong>
10     {
11         protected new readonly RawLink<ulong>* Links;
12         protected new readonly LinksHeader<ulong>* Header;
13
14         protected UInt64LinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<ulong>
15             ↳ constants, RawLink<ulong>* links, LinksHeader<ulong>* header)
16             : base(constants, (byte*)links, (byte*)header)
17         {
18             Links = links;
19             Header = header;
20
21             [MethodImpl(MethodImplOptions.AggressiveInlining)]
22             protected override ulong GetZero() => OUL;
23
24             [MethodImpl(MethodImplOptions.AggressiveInlining)]
25             protected override bool EqualToZero(ulong value) => value == OUL;
26
27             [MethodImpl(MethodImplOptions.AggressiveInlining)]
28             protected override bool AreEqual(ulong first, ulong second) => first == second;
29
30             [MethodImpl(MethodImplOptions.AggressiveInlining)]
31             protected override bool GreaterThanZero(ulong value) => value > OUL;
32
33             [MethodImpl(MethodImplOptions.AggressiveInlining)]
34             protected override bool GreaterThan(ulong first, ulong second) => first > second;
35
36             [MethodImpl(MethodImplOptions.AggressiveInlining)]
37             protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
38
39             [MethodImpl(MethodImplOptions.AggressiveInlining)]
40             protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
41                 ↳ always true for ulong
42
43             [MethodImpl(MethodImplOptions.AggressiveInlining)]
44             protected override bool LessOrEqualThanZero(ulong value) => value == OUL; // value is
45                 ↳ always >= 0 for ulong
46
47             [MethodImpl(MethodImplOptions.AggressiveInlining)]
48             protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
49
50             [MethodImpl(MethodImplOptions.AggressiveInlining)]
51             protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
52                 ↳ for ulong
53
54             [MethodImpl(MethodImplOptions.AggressiveInlining)]
55             protected override bool LessThan(ulong first, ulong second) => first < second;
56
57             [MethodImpl(MethodImplOptions.AggressiveInlining)]
58             protected override ulong Increment(ulong value) => ++value;
59
60             [MethodImpl(MethodImplOptions.AggressiveInlining)]
61             protected override ulong Decrement(ulong value) => --value;
62
63             [MethodImpl(MethodImplOptions.AggressiveInlining)]
64             protected override ulong Add(ulong first, ulong second) => first + second;
65
66             [MethodImpl(MethodImplOptions.AggressiveInlining)]
67             protected override ulong Subtract(ulong first, ulong second) => first - second;
68
69             [MethodImpl(MethodImplOptions.AggressiveInlining)]
70             protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
71             {
72                 ref var firstLink = ref Links[first];
73                 ref var secondLink = ref Links[second];
74                 return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
75                     ↳ secondLink.Source, secondLink.Target);
76             }
77         }
78     }
79 }
```

```

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,
79             ↪ secondLink.Source, secondLink.Target);
80     }
81
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
87 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.United.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.United.Specific
7  {
8      public unsafe abstract class UInt64LinksSizeBalancedTreeMethodsBase :
9          ↪ LinksSizeBalancedTreeMethodsBase<ulong>
10     {
11         protected new readonly RawLink<ulong>* Links;
12         protected new readonly LinksHeader<ulong>* Header;
13
14         protected UInt64LinksSizeBalancedTreeMethodsBase(LinksConstants<ulong> constants,
15             ↪ RawLink<ulong>* links, LinksHeader<ulong>* header)
16             : base(constants, (byte*)links, (byte*)header)
17         {
18             Links = links;
19             Header = header;
20         }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetZero() => OUL;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override bool EqualToZero(ulong value) => value == OUL;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override bool AreEqual(ulong first, ulong second) => first == second;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override bool GreaterThanZero(ulong value) => value > OUL;
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected override bool GreaterThan(ulong first, ulong second) => first > second;
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
42             ↪ always true for ulong
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override bool LessOrEqualThanZero(ulong value) => value == OUL; // value is
46             ↪ always >= 0 for ulong
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
53             ↪ for ulong
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override bool LessThan(ulong first, ulong second) => first < second;
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         protected override ulong Increment(ulong value) => ++value;
60     }

```



```

56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 protected override ulong Decrement(ulong value) => --value;
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected override ulong Add(ulong first, ulong second) => first + second;
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override ulong Subtract(ulong first, ulong second) => first - second;
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override bool FirstIsToLeftOfSecond(ulong first, ulong second)
67 {
68     ref var firstLink = ref Links[first];
69     ref var secondLink = ref Links[second];
70     return FirstIsToLeftOfSecond(firstLink.Source, firstLink.Target,
71         ↪ secondLink.Source, secondLink.Target);
72 }
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
76 {
77     ref var firstLink = ref Links[first];
78     ref var secondLink = ref Links[second];
79     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
80         ↪ secondLink.Source, secondLink.Target);
81 }
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
88 }

```

1.101 ./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)]

```

```

36     protected override bool GetLeftIsChild(ulong node) =>
37         ↳ GetLeftIsChildValue(Links[node].SizeAsSource);
38
39     //[MethodImpl(MethodImplOptions.AggressiveInlining)]
40     //protected override bool GetLeftIsChild(ulong node) => IsChild(node, GetLeft(node));
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override void SetLeftIsChild(ulong node, bool value) =>
44         ↳ SetLeftIsChildValue(ref Links[node].SizeAsSource, value);
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override bool GetRightIsChild(ulong node) =>
48         ↳ GetRightIsChildValue(Links[node].SizeAsSource);
49
50     //[MethodImpl(MethodImplOptions.AggressiveInlining)]
51     //protected override bool GetRightIsChild(ulong node) => IsChild(node, GetRight(node));
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override void SetRightIsChild(ulong node, bool value) =>
55         ↳ SetRightIsChildValue(ref Links[node].SizeAsSource, value);
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override sbyte GetBalance(ulong node) =>
59         ↳ GetBalanceValue(Links[node].SizeAsSource);
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
63         ↳ Links[node].SizeAsSource, value);
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override ulong GetTreeRoot() => Header->RootAsSource;
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     protected override bool FirstIsToTheLeftOfSecond(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 bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
79         ↳ ulong secondSource, ulong secondTarget)
80         => firstSource > secondSource || (firstSource == secondSource && firstTarget >
81         ↳ secondTarget);
82
83     [MethodImpl(MethodImplOptions.AggressiveInlining)]
84     protected override void ClearNode(ulong node)
85     {
86         ref var link = ref Links[node];
87         link.LeftAsSource = OUL;
88         link.RightAsSource = OUL;
89         link.SizeAsSource = OUL;
90     }
91 }

```

1.102 ./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;

```

```

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

```

1.103 ./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     }
29 }

```

```

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

```

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

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.United.Specific
6  {
7      public unsafe class UInt64LinksTargetsAvlBalancedTreeMethods :
        ↳ UInt64LinksAvlBalancedTreeMethodsBase
8      {
9          public UInt64LinksTargetsAvlBalancedTreeMethods(LinksConstants<ulong> constants,
        ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
        ↳ { }
10
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         protected override ref ulong GetLeftReference(ulong node) => ref
        ↳ Links[node].LeftAsTarget;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetRightReference(ulong node) => ref
        ↳ Links[node].RightAsTarget;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
        ↳ left;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
        ↳ right;

```

```

28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsTarget);
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
33         ↳ Links[node].SizeAsTarget, size);
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override bool GetLeftIsChild(ulong node) =>
37         ↳ GetLeftIsChildValue(Links[node].SizeAsTarget);
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected override void SetLeftIsChild(ulong node, bool value) =>
41         ↳ SetLeftIsChildValue(ref Links[node].SizeAsTarget, value);
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override bool GetRightIsChild(ulong node) =>
45         ↳ GetRightIsChildValue(Links[node].SizeAsTarget);
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override void SetRightIsChild(ulong node, bool value) =>
49         ↳ SetRightIsChildValue(ref Links[node].SizeAsTarget, value);
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override sbyte GetBalance(ulong node) =>
53         ↳ GetBalanceValue(Links[node].SizeAsTarget);
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
57         ↳ Links[node].SizeAsTarget, value);
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override ulong GetTreeRoot() => Header->RootAsTarget;
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
67         ↳ ulong secondSource, ulong secondTarget)
68         => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
69         ↳ secondSource);
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
73         ↳ ulong secondSource, ulong secondTarget)
74         => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
75         ↳ secondSource);
76
77     [MethodImpl(MethodImplOptions.AggressiveInlining)]
78     protected override void ClearNode(ulong node)
79     {
80         ref var link = ref Links[node];
81         link.LeftAsTarget = OUL;
82         link.RightAsTarget = OUL;
83         link.SizeAsTarget = OUL;
84     }
85 }
86 }

```

1.105 ./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 :
8          ↳ UInt64LinksRecursionlessSizeBalancedTreeMethodsBase
9      {
10         public UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods(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].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 void SetLeftReference(ulong node, ref ulong value) =>
24             ↳ SetLeftReferenceValue(ref Links[node].LeftAsTarget, value);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override void SetRightReference(ulong node, ref ulong value) =>
28             ↳ SetRightReferenceValue(ref Links[node].RightAsTarget, value);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override void ClearNode(ulong node)
32         {
33             ref var link = ref Links[node];
34             link.LeftAsTarget = OUL;
35             link.RightAsTarget = OUL;
36             link.SizeAsTarget = OUL;
37         }
38     }
39 }

```

```

13     [MethodImpl(MethodImplOptions.AggressiveInlining)]
14     protected override ref ulong GetRightReference(ulong node) => ref
15     ↪ 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 =
25     ↪ left;
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
29     ↪ right;
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
36     ↪ size;
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override ulong GetTreeRoot() => Header->RootAsTarget;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override bool FirstIsToTheLeftOfSecond(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 bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
52     ↪ ulong secondSource, ulong secondTarget)
53     => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
54     ↪ secondSource);
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected override void ClearNode(ulong node)
58     {
59         ref var link = ref Links[node];
60         link.LeftAsTarget = OUL;
61         link.RightAsTarget = OUL;
62         link.SizeAsTarget = OUL;
63     }
64 }

```

1.106 ./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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
57         ↪ ulong secondSource, ulong secondTarget)
58         => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
59         ↪ secondSource);
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         protected override void ClearNode(ulong node)
63         {
64             ref var link = ref Links[node];
65             link.LeftAsTarget = OUL;
66             link.RightAsTarget = OUL;
67             link.SizeAsTarget = OUL;
68         }
69     }
70 }

```

```

19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
24         ↳ left;
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
28         ↳ right;
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
35         ↳ size;
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override ulong GetTreeRoot() => Header->RootAsTarget;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override bool FirstIsToLeftOfSecond(ulong firstSource, ulong firstTarget,
45         ↳ ulong secondSource, ulong secondTarget)
46         => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
47             ↳ secondSource);
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
51         ↳ ulong secondSource, ulong secondTarget)
52         => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
53             ↳ secondSource);
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override void ClearNode(ulong node)
57     {
58         ref var link = ref Links[node];
59         link.LeftAsTarget = OUL;
60         link.RightAsTarget = OUL;
61         link.SizeAsTarget = OUL;
62     }
63 }

```

1.107 ./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         /// <para>Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
31         ↳ минимальным шагом расширения базы данных.

```

```

27  /// </summary>
28  /// <param name="address">Полный путь к файлу базы данных.</param>
29  /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
    ↳ байтах.</param>
30  [MethodImpl(MethodImplOptions.AggressiveInlining)]
31  public UInt64UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
    ↳ FileMappedResizableDirectMemory(address, memoryReservationStep),
    ↳ memoryReservationStep) { }
32
33  [MethodImpl(MethodImplOptions.AggressiveInlining)]
34  public UInt64UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
    ↳ DefaultLinksSizeStep) { }
35
36  [MethodImpl(MethodImplOptions.AggressiveInlining)]
37  public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
    ↳ memoryReservationStep) : this(memory, memoryReservationStep,
    ↳ Default<LinksConstants<ulong>>.Instance, IndexTreeType.Default) { }
38
39  [MethodImpl(MethodImplOptions.AggressiveInlining)]
40  public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
    ↳ memoryReservationStep, LinksConstants<ulong> constants, IndexTreeType indexTreeType)
    ↳ : base(memory, memoryReservationStep, constants)
41  {
42      if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
43      {
44          _createSourceTreeMethods = () => new
            ↳ UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
45          _createTargetTreeMethods = () => new
            ↳ UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
46      }
47      else if (indexTreeType == IndexTreeType.SizeBalancedTree)
48      {
49          _createSourceTreeMethods = () => new
            ↳ UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
50          _createTargetTreeMethods = () => new
            ↳ UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
51      }
52      else
53      {
54          _createSourceTreeMethods = () => new
            ↳ UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods(Constants, _links,
            ↳ _header);
55          _createTargetTreeMethods = () => new
            ↳ UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods(Constants, _links,
            ↳ _header);
56      }
57      Init(memory, memoryReservationStep);
58  }
59
60  [MethodImpl(MethodImplOptions.AggressiveInlining)]
61  protected override void SetPointers(IResizableDirectMemory memory)
62  {
63      _header = (LinksHeader<ulong>*)memory.Pointer;
64      _links = (RawLink<ulong>*)memory.Pointer;
65      SourcesTreeMethods = _createSourceTreeMethods();
66      TargetsTreeMethods = _createTargetTreeMethods();
67      UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
68  }
69
70  [MethodImpl(MethodImplOptions.AggressiveInlining)]
71  protected override void ResetPointers()
72  {
73      base.ResetPointers();
74      _links = null;
75      _header = null;
76  }
77
78  [MethodImpl(MethodImplOptions.AggressiveInlining)]
79  protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
80
81  [MethodImpl(MethodImplOptions.AggressiveInlining)]
82  protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
    ↳ _links[linkIndex];
83
84  [MethodImpl(MethodImplOptions.AggressiveInlining)]
85  protected override bool AreEqual(ulong first, ulong second) => first == second;
86

```



```

87     [MethodImpl(MethodImplOptions.AggressiveInlining)]
88     protected override bool LessThan(ulong first, ulong second) => first < second;
89
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     protected override bool GreaterThan(ulong first, ulong second) => first > second;
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
98
99     [MethodImpl(MethodImplOptions.AggressiveInlining)]
100    protected override ulong GetZero() => 0UL;
101
102    [MethodImpl(MethodImplOptions.AggressiveInlining)]
103    protected override ulong GetOne() => 1UL;
104
105    [MethodImpl(MethodImplOptions.AggressiveInlining)]
106    protected override long ConvertToInt64(ulong value) => (long)value;
107
108    [MethodImpl(MethodImplOptions.AggressiveInlining)]
109    protected override ulong ConvertToAddress(long value) => (ulong)value;
110
111    [MethodImpl(MethodImplOptions.AggressiveInlining)]
112    protected override ulong Add(ulong first, ulong second) => first + second;
113
114    [MethodImpl(MethodImplOptions.AggressiveInlining)]
115    protected override ulong Subtract(ulong first, ulong second) => first - second;
116
117    [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 }

```

1.108 ./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 }

```

1.109 ./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,
10         ↪ TLink>
11      {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↪ EqualityComparer<TLink>.Default;

```

```

12     [MethodImpl(MethodImplOptions.AggressiveInlining)]
13     public PropertiesOperator(ILinks<TLink> links) : base(links) { }
14
15     [MethodImpl(MethodImplOptions.AggressiveInlining)]
16     public TLink GetValue(TLink @object, TLink property)
17     {
18         var links = _links;
19         var objectProperty = links.SearchOrDefault(@object, property);
20         if (_equalityComparer.Equals(objectProperty, default))
21         {
22             return default;
23         }
24         var constants = links.Constants;
25         var any = constants.Any;
26         var query = new Link<TLink>(any, objectProperty, any);
27         var valueLink = links.SingleOrDefault(query);
28         if (valueLink == null)
29         {
30             return default;
31         }
32         return links.GetTarget(valueLink[constants.IndexPart]);
33     }
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     public void SetValue(TLink @object, TLink property, TLink value)
37     {
38         var links = _links;
39         var objectProperty = links.GetOrCreate(@object, property);
40         links.DeleteMany(links.AllIndices(links.Constants.Any, objectProperty));
41         links.GetOrCreate(objectProperty, value);
42     }
43 }
44
45 }

```

1.110 ./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;

```

```

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

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Collections.Stacks;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Stacks
8  {
9      public class Stack<TLink> : LinksOperatorBase<TLink>, IStack<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13
14         private readonly TLink _stack;
15
16         public bool IsEmpty
17         {
18             [MethodImpl(MethodImplOptions.AggressiveInlining)]
19             get => _equalityComparer.Equals(Peek(), _stack);
20         }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public Stack(ILinks<TLink> links, TLink stack) : base(links) => _stack = stack;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         private TLink GetStackMarker() => _links.GetSource(_stack);
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         private TLink GetTop() => _links.GetTarget(_stack);
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public TLink Peek() => _links.GetTarget(GetTop());
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public TLink Pop()
36         {
37             var element = Peek();
38             if (!_equalityComparer.Equals(element, _stack))
39             {
40                 var top = GetTop();
41                 var previousTop = _links.GetSource(top);
42                 _links.Update(_stack, GetStackMarker(), previousTop);

```

```

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 }

```

1.112 ./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 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Data.Doublets;
5 using Platform.Threading.Synchronization;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets
10 {
11     /// <remarks>
12     /// TODO: Autogeneration of synchronized wrapper (decorator).
13     /// TODO: Try to unfold code of each method using IL generation for performance improvements.
14     /// TODO: Or even to unfold multiple layers of implementations.
15     /// </remarks>
16     public class SynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress>
17     {
18         public LinksConstants<TLinkAddress> Constants
19         {
20             [MethodImpl(MethodImplOptions.AggressiveInlining)]
21             get;
22         }
23
24         public ISynchronization SyncRoot
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28         }
29
30         public ILinks<TLinkAddress> Sync
31         {
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             get;
34         }
35
36         public ILinks<TLinkAddress> Unsync
37         {
38             [MethodImpl(MethodImplOptions.AggressiveInlining)]
39             get;
40         }
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
44             ↪ ReaderWriterLockSynchronization(), links) { }
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
48         {
49             SyncRoot = synchronization;

```

```

49     Sync = this;
50     Unsync = links;
51     Constants = links.Constants;
52 }
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 public TLinkAddress Count(ICollection<TLinkAddress> restriction) =>
56     ↳ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 public TLinkAddress Each(Func<ICollection<TLinkAddress>, TLinkAddress> handler,
60     ↳ ICollection<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
61     ↳ restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 public TLinkAddress Create(ICollection<TLinkAddress> restrictions) =>
65     ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 public TLinkAddress Update(ICollection<TLinkAddress> restrictions, ICollection<TLinkAddress>
69     ↳ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
70     ↳ Unsync.Update);
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 public void Delete(ICollection<TLinkAddress> restrictions) =>
74     ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
75
76 //public T Trigger(ICollection<T> restriction, Func<ICollection<T>, ICollection<T>, T> matchedHandler,
77 //    ↳ ICollection<T> substitution, Func<ICollection<T>, ICollection<T>, T> substitutedHandler)
78 //{{
79 //    if (restriction != null && substitution != null &&
80 //        ↳ !substitution.EqualTo(restriction))
81 //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
82 //        ↳ substitution, substitutedHandler, Unsync.Trigger);
83 //
84 //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
85 //        ↳ substitutedHandler, Unsync.Trigger);
86 //}}
87 }
88 }

```

1.114 ./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
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets
10 {
11     public static class UInt64LinksExtensions
12     {
13         public static readonly LinksConstants<ulong> Constants =
14             ↳ Default<LinksConstants<ulong>>.Instance;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public static bool AnyLinkIsAny(this ICollection<ulong> links, params ulong[] sequence)
18         {
19             if (sequence == null)
20             {
21                 return false;
22             }
23             var constants = links.Constants;
24             for (var i = 0; i < sequence.Length; i++)
25             {
26                 if (sequence[i] == constants.Any)
27                 {
28                     return true;
29                 }
30             }
31             return false;
32         }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public static string FormatStructure(this ICollection<ulong> links, ulong linkIndex,
36             ↳ Func<Link<ulong>, bool> isElement, bool renderIndex = false, bool renderDebug =
37             ↳ false)

```

```

35 {
36     var sb = new StringBuilder();
37     var visited = new HashSet<ulong>();
38     links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
        ↪ innerSb.Append(link.Index), renderIndex, renderDebug);
39     return sb.ToString();
40 }
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
    ↪ Func<Link<ulong>, bool> isElement, Action<StringBuilder, Link<ulong>> appendElement,
    ↪ bool renderIndex = false, bool renderDebug = false)
44 {
45     var sb = new StringBuilder();
46     var visited = new HashSet<ulong>();
47     links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
        ↪ renderDebug);
48     return sb.ToString();
49 }
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 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)
53 {
54     if (sb == null)
55     {
56         throw new ArgumentNullException(nameof(sb));
57     }
58     if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
        ↪ Constants.Itself)
59     {
60         return;
61     }
62     if (links.Exists(linkIndex))
63     {
64         if (visited.Add(linkIndex))
65         {
66             sb.Append('(');
67             var link = new Link<ulong>(links.GetLink(linkIndex));
68             if (renderIndex)
69             {
70                 sb.Append(link.Index);
71                 sb.Append(':');
72             }
73             if (link.Source == link.Index)
74             {
75                 sb.Append(link.Index);
76             }
77             else
78             {
79                 var source = new Link<ulong>(links.GetLink(link.Source));
80                 if (isElement(source))
81                 {
82                     appendElement(sb, source);
83                 }
84                 else
85                 {
86                     links.AppendStructure(sb, visited, source.Index, isElement,
                        ↪ appendElement, renderIndex);
87                 }
88             }
89             sb.Append(' ');
90             if (link.Target == link.Index)
91             {
92                 sb.Append(link.Index);
93             }
94             else
95             {
96                 var target = new Link<ulong>(links.GetLink(link.Target));
97                 if (isElement(target))
98                 {
99                     appendElement(sb, target);
100                 }
101                 else
102                 {

```

```

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

```

1.115 ./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 //-V3073
20     {
21         /// <remarks>
22         /// Альтернативные варианты хранения трансформации (элемента транзакции):
23         ///
24         /// private enum TransitionType
25         /// {
26         ///     Creation,
27         ///     UpdateOf,
28         ///     UpdateTo,
29         ///     Deletion
30         /// }
31         ///
32         /// private struct Transition
33         /// {
34         ///     public ulong TransactionId;
35         ///     public UniqueTimestamp Timestamp;
36         ///     public TransactionItemType Type;
37         ///     public Link Source;
38         ///     public Link Linker;
39         ///     public Link Target;
40         /// }
41         ///
42         /// Или
43         ///
44         /// public struct TransitionHeader
45         /// {
46         ///     public ulong TransactionIdCombined;
47         ///     public ulong TimestampCombined;
48         ///
49         ///     public ulong TransactionId
50         ///     {
51         ///         get

```

```

52     /// {
53     ///     return (ulong) mask & TransactionIdCombined;
54     /// }
55     /// }
56     ///
57     public UniqueTimestamp Timestamp
58     {
59         get
60         {
61             return (UniqueTimestamp)mask & TransactionIdCombined;
62         }
63     }
64     ///
65     public TransactionItemType Type
66     {
67         get
68         {
69             // Использовать по одному биту из TransactionId и Timestamp,
70             // для значения в 2 бита, которое представляет тип операции
71             throw new NotImplementedException();
72         }
73     }
74 }
75 ///
76 private struct Transition
77 {
78     public TransitionHeader Header;
79     public Link Source;
80     public Link Linker;
81     public Link Target;
82 }
83 ///
84 </remarks>
85 public struct Transition : IEquatable<Transition>
86 {
87     public static readonly long Size = Structure<Transition>.Size;
88
89     public readonly ulong TransactionId;
90     public readonly Link<ulong> Before;
91     public readonly Link<ulong> After;
92     public readonly Timestamp Timestamp;
93
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;

```



```

121     [MethodImpl(MethodImplOptions.AggressiveInlining)]
122     public static bool operator ==(Transition left, Transition right) =>
123         ↪ left.Equals(right);
124
125     [MethodImpl(MethodImplOptions.AggressiveInlining)]
126     public static bool operator !=(Transition left, Transition right) => !(left ==
127         ↪ right);
128 }
129
130 /// <remarks>
131 /// Другие варианты реализации транзакций (атомарности):
132 /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
133     ↪ Target)) и индексов.
134 /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
135     ↪ потребуется решить вопрос
136     со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
137     ↪ пересечениями идентификаторов.
138 ///
139 /// Где хранить промежуточный список транзакций?
140 ///
141 /// В оперативной памяти:
142 /// Минусы:
143 /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
144     так как нужно отдельно выделять память под список трансформаций.
145 /// 2. Выделенной оперативной памяти может не хватить, в том случае,
146     если транзакция использует слишком много трансформаций.
147     -> Можно использовать жёсткий диск для слишком длинных транзакций.
148     -> Максимальный размер списка трансформаций можно ограничить / задать
149     ↪ константой.
150 /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
151     ↪ создавая задержку.
152 ///
153 /// На жёстком диске:
154 /// Минусы:
155 /// 1. Длительный отклик, на запись каждой трансформации.
156 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
157     -> Это может решаться упаковкой/исключением дублирующих операций.
158     -> Также это может решаться тем, что короткие транзакции вообще
159     ↪ не будут записываться в случае отката.
160 /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
161     ↪ операции (трансформации)
162     ↪ будут записаны в лог.
163 ///
164 /// </remarks>
165 public class Transaction : DisposableBase
166 {
167     private readonly Queue<Transition> _transitions;
168     private readonly UInt64LinksTransactionsLayer _layer;
169     public bool IsCommitted { get; private set; }
170     public bool IsReverted { get; private set; }
171
172     [MethodImpl(MethodImplOptions.AggressiveInlining)]
173     public Transaction(UInt64LinksTransactionsLayer layer)
174     {
175         _layer = layer;
176         if (_layer._currentTransactionId != 0)
177         {
178             throw new NotSupportedException("Nested transactions not supported.");
179         }
180         IsCommitted = false;
181         IsReverted = false;
182         _transitions = new Queue<Transition>();
183         SetCurrentTransaction(layer, this);
184     }
185
186     [MethodImpl(MethodImplOptions.AggressiveInlining)]
187     public void Commit()
188     {
189         EnsureTransactionAllowsWriteOperations(this);
190         while (_transitions.Count > 0)
191         {
192             var transition = _transitions.Dequeue();
193             _layer._transitions.Enqueue(transition);
194         }
195         _layer._lastCommittedTransactionId = _layer._currentTransactionId;
196         IsCommitted = true;
197     }
198 }

```

```

191 [MethodImpl(MethodImplOptions.AggressiveInlining)]
192 private void Revert()
193 {
194     EnsureTransactionAllowsWriteOperations(this);
195     var transitionsToRevert = new Transition[_transitions.Count];
196     _transitions.CopyTo(transitionsToRevert, 0);
197     for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
198     {
199         _layer.RevertTransition(transitionsToRevert[i]);
200     }
201     IsReverted = true;
202 }
203
204 [MethodImpl(MethodImplOptions.AggressiveInlining)]
205 public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
    ↳ Transaction transaction)
206 {
207     layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
208     layer._currentTransactionTransitions = transaction._transitions;
209     layer._currentTransaction = transaction;
210 }
211
212 [MethodImpl(MethodImplOptions.AggressiveInlining)]
213 public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
214 {
215     if (transaction.IsReverted)
216     {
217         throw new InvalidOperationException("Transation is reverted.");
218     }
219     if (transaction.IsCommitted)
220     {
221         throw new InvalidOperationException("Transation is committed.");
222     }
223 }
224
225 [MethodImpl(MethodImplOptions.AggressiveInlining)]
226 protected override void Dispose(bool manual, bool wasDisposed)
227 {
228     if (!wasDisposed && _layer != null && !_layer.Disposable.IsDisposed)
229     {
230         if (!IsCommitted && !IsReverted)
231         {
232             Revert();
233         }
234         _layer.ResetCurrentTransation();
235     }
236 }
237
238
239 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
240
241 private readonly string _logAddress;
242 private readonly FileStream _log;
243 private readonly Queue<Transition> _transitions;
244 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
245 private Task _transitionsPusher;
246 private Transition _lastCommittedTransition;
247 private ulong _currentTransactionId;
248 private Queue<Transition> _currentTransactionTransitions;
249 private Transaction _currentTransaction;
250 private ulong _lastCommittedTransactionId;
251
252 [MethodImpl(MethodImplOptions.AggressiveInlining)]
253 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
254     : base(links)
255 {
256     if (string.IsNullOrEmpty(logAddress))
257     {
258         throw new ArgumentNullException(nameof(logAddress));
259     }
260     // В первой строке файла хранится последняя закоммиченную транзакцию.
261     // При запуске это используется для проверки удачного закрытия файла лога.
262     // In the first line of the file the last committed transaction is stored.
263     // On startup, this is used to check that the log file is successfully closed.
264     var lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
265     var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
266     if (!lastCommittedTransition.Equals(lastWrittenTransition))
267     {
268         Dispose();

```

```

269         throw new NotSupportedException("Database is damaged, autorecovery is not
270         ↪ supported yet.");
271     }
272     if (lastCommittedTransition == default)
273     {
274         FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
275     }
276     _lastCommittedTransition = lastCommittedTransition;
277     // TODO: Think about a better way to calculate or store this value
278     var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
279     _lastCommittedTransactionId = allTransitions.Length > 0 ? allTransitions.Max(x =>
280     ↪ x.TransactionId) : 0;
281     _uniqueTimestampFactory = new UniqueTimestampFactory();
282     _logAddress = logAddress;
283     _log = FileHelpers.Append(logAddress);
284     _transitions = new Queue<Transition>();
285     _transitionsPusher = new Task(TransitionsPusher);
286     _transitionsPusher.Start();
287 }
288
289 [MethodImpl(MethodImplOptions.AggressiveInlining)]
290 public IList<ulong> GetLinkValue(ulong link) => _links.GetLink(link);
291
292 [MethodImpl(MethodImplOptions.AggressiveInlining)]
293 public override ulong Create(IList<ulong> restrictions)
294 {
295     var createdLinkIndex = _links.Create();
296     var createdLink = new Link<ulong>(_links.GetLink(createdLinkIndex));
297     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
298     ↪ default, createdLink));
299     return createdLinkIndex;
300 }
301
302 [MethodImpl(MethodImplOptions.AggressiveInlining)]
303 public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
304 {
305     var linkIndex = restrictions[_constants.IndexPart];
306     var beforeLink = new Link<ulong>(_links.GetLink(linkIndex));
307     linkIndex = _links.Update(restrictions, substitution);
308     var afterLink = new Link<ulong>(_links.GetLink(linkIndex));
309     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
310     ↪ beforeLink, afterLink));
311     return linkIndex;
312 }
313
314 [MethodImpl(MethodImplOptions.AggressiveInlining)]
315 public override void Delete(IList<ulong> restrictions)
316 {
317     var link = restrictions[_constants.IndexPart];
318     var deletedLink = new Link<ulong>(_links.GetLink(link));
319     _links.Delete(link);
320     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
321     ↪ deletedLink, default));
322 }
323
324 [MethodImpl(MethodImplOptions.AggressiveInlining)]
325 private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
326     ↪ _transitions;
327
328 [MethodImpl(MethodImplOptions.AggressiveInlining)]
329 private void CommitTransition(Transition transition)
330 {
331     if (_currentTransaction != null)
332     {
333         Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
334     }
335     var transitions = GetCurrentTransitions();
336     transitions.Enqueue(transition);
337 }
338
339 [MethodImpl(MethodImplOptions.AggressiveInlining)]
340 private void RevertTransition(Transition transition)
341 {
342     if (transition.After.IsNull()) // Revert Deletion with Creation
343     {
344         _links.Create();
345     }
346     else if (transition.Before.IsNull()) // Revert Creation with Deletion

```

```

341     {
342         _links.Delete(transition.After.Index);
343     }
344     else // Revert Update
345     {
346         _links.Update(new[] { transition.After.Index, transition.Before.Source,
347             ↪ transition.Before.Target });
348     }
349 }
350 [MethodImpl(MethodImplOptions.AggressiveInlining)]
351 private void ResetCurrentTransation()
352 {
353     _currentTransactionId = 0;
354     _currentTransactionTransitions = null;
355     _currentTransaction = null;
356 }
357 [MethodImpl(MethodImplOptions.AggressiveInlining)]
358 private void PushTransitions()
359 {
360     if (_log == null || _transitions == null)
361     {
362         return;
363     }
364     for (var i = 0; i < _transitions.Count; i++)
365     {
366         var transition = _transitions.Dequeue();
367
368         _log.Write(transition);
369         _lastCommittedTransition = transition;
370     }
371 }
372 }
373 [MethodImpl(MethodImplOptions.AggressiveInlining)]
374 private void TransitionsPusher()
375 {
376     while (!Disposable.IsDisposed && _transitionsPusher != null)
377     {
378         Thread.Sleep(DefaultPushDelay);
379         PushTransitions();
380     }
381 }
382 }
383 [MethodImpl(MethodImplOptions.AggressiveInlining)]
384 public Transaction BeginTransaction() => new Transaction(this);
385 [MethodImpl(MethodImplOptions.AggressiveInlining)]
386 private void DisposeTransitions()
387 {
388     try
389     {
390         {
391             var pusher = _transitionsPusher;
392             if (pusher != null)
393             {
394                 _transitionsPusher = null;
395                 pusher.Wait();
396             }
397             if (_transitions != null)
398             {
399                 PushTransitions();
400             }
401             _log.DisposeIfPossible();
402             FileHelpers.WriteFirst(_logAddress, _lastCommittedTransition);
403         }
404         catch (Exception ex)
405         {
406             ex.Ignore();
407         }
408     }
409 }
410 #region DisposalBase
411 [MethodImpl(MethodImplOptions.AggressiveInlining)]
412 protected override void Dispose(bool manual, bool wasDisposed)
413 {
414     if (!wasDisposed)
415     {
416         DisposeTransitions();
417     }
418 }

```

```

419     }
420     base.Dispose(manual, wasDisposed);
421 }
422
423 #endregion
424 }
425 }

```

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

```

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

```

1.117 ./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 }

```

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

```

1  using System.IO;
2  using Xunit;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using Platform.Data.Doublets.Memory.United.Specific;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public static class ResizableDirectMemoryLinksTests
10     {
11         private static readonly LinksConstants<ulong> _constants =
12             ↪ Default<LinksConstants<ulong>>.Instance;
13
14         [Fact]
15         public static void BasicFileMappedMemoryTest()
16         {
17             var tempFilename = Path.GetTempFileName();
18             using (var memoryAdapter = new UInt64UnitedMemoryLinks(tempFilename))
19             {
20                 memoryAdapter.TestBasicMemoryOperations();
21             }
22             File.Delete(tempFilename);
23
24             [Fact]
25             public static void BasicHeapMemoryTest()
26             {
27                 using (var memory = new
28                     ↪ HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
29                 using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
30                     ↪ UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
31                 {
32                     memoryAdapter.TestBasicMemoryOperations();
33                 }
34
35                 private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
36                 {
37                     var link = memoryAdapter.Create();
38                     memoryAdapter.Delete(link);
39                 }
40
41                 [Fact]
42                 public static void NonexistentReferencesHeapMemoryTest()
43                 {
44                     using (var memory = new
45                         ↪ HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
46                     using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
47                         ↪ UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
48                     {
49                         memoryAdapter.TestNonexistentReferences();
50                     }
51
52                     private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
53                     {
54                         var link = memoryAdapter.Create();
55                         memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
56                         var resultLink = _constants.Null;
57                         memoryAdapter.Each(foundLink =>
58                         {
59                             resultLink = foundLink[_constants.IndexPart];
60                             return _constants.Break;
61                         }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
62                         Assert.True(resultLink == link);
63                         Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
64                         memoryAdapter.Delete(link);
65                     }
66                 }
67             }
68         }
69     }
70 }

```

1.119 ./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     }

```

1.120 ./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
        ↳ MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
        ↳ implementation of tree cuts out 5 bits from the address space.
33     Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
        ↳ stMultipleRandomCreationsAndDeletions(100));
34     Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
        ↳ MultipleRandomCreationsAndDeletions(100));
35     Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
        ↳ tMultipleRandomCreationsAndDeletions(100));
36 }
37
38 private static void Using<TLink>(Action<ILinks<TLink>> action)
39 {
40     using (var dataMemory = new HeapResizableDirectMemory())
41     using (var indexMemory = new HeapResizableDirectMemory())
42     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 contants = 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,
        ↳ SplitMemoryLinks<TLink>.DefaultLinksSizeStep, contants))
54     {
55         action(memory);
56     }
57 }
58 }
59 }

```

1.121 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt32LinksTests.cs

```

1  using System;
2  using Xunit;
3  using Platform.Memory;
4  using Platform.Data.Doublets.Memory.Split.Specific;
5  using TLink = System.UInt32;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public unsafe static class SplitMemoryUInt32LinksTests
10     {
11         [Fact]
12         public static void CRUDTest()
13         {
14             Using(links => links.TestCRUDOperations());
15         }
16
17         [Fact]
18         public static void RawNumbersCRUDTest()
19         {
20             UsingWithExternalReferences(links => links.TestRawNumbersCRUDOperations());
21         }
22
23         [Fact]
24         public static void MultipleRandomCreationsAndDeletionsTest()
25         {
26             Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultip
        ↳ leRandomCreationsAndDeletions(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 }

```

1.122 ./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     }

```

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

```

1.124 ./csharp/Platform.Data.Doublets.Tests/UInt64LinksExtensionsTests.cs

```

1 using Platform.Data.Doublets.Memory;
2 using Platform.Data.Doublets.Memory.United.Generic;
3 using Platform.Data.Numbers.Raw;
4 using Platform.Memory;
5 using Platform.Numbers;
6 using Xunit;
7 using Xunit.Abstractions;
8 using TLink = System.UInt64;
9
10 namespace Platform.Data.Doublets.Tests
11 {
12     public class UInt64LinksExtensionsTests
13     {
14         public static ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new
            ↪ Platform.IO.TemporaryFile());
15
16         public static ILinks<TLink> CreateLinks<TLink>(string dataDBFilename)
17         {
18             var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
                ↪ true);
19             return new UnitedMemoryLinks<TLink>(new
                ↪ FileMappedResizableDirectMemory(dataDBFilename),
                ↪ UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
                ↪ IndexTreeType.Default);
20         }
21         [Fact]
22         public void FormatStructureWithExternalReferenceTest()
23         {
24             ILinks<TLink> links = CreateLinks();
25             TLink zero = default;

```

```

26     var one = Arithmetic.Increment(zero);
27     var markerIndex = one;
28     var meaningRoot = links.GetOrCreate(markerIndex, markerIndex);
29     var numberMarker = links.GetOrCreate(meaningRoot, Arithmetic.Increment(ref
        ↳ markerIndex));
30     AddressToRawNumberConverter<TLink> addressToNumberConverter = new();
31     var numberAddress = addressToNumberConverter.Convert(1);
32     var numberLink = links.GetOrCreate(numberMarker, numberAddress);
33     var linkNotation = links.FormatStructure(numberLink, link => link.IsFullPoint(),
        ↳ true);
34     Assert.Equal("(3:(2:1 2) 18446744073709551615)", linkNotation);
35 }
36 }
37 }

```

1.125 ./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().TestMultip
                ↳ leRandomCreationsAndDeletions(100));
29         }
30
31         private static void Using(Action<ILinks<TLink>> action)
32         {
33             using (var scope = new Scope<Types<HeapResizableDirectMemory,
                ↳ UInt32UnitedMemoryLinks>>())
34             {
35                 action(scope.Use<ILinks<TLink>>());
36             }
37         }
38     }
39 }

```

1.126 ./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt64LinksTests.cs

```

1  using System;
2  using Xunit;
3  using Platform.Reflection;
4  using Platform.Memory;
5  using Platform.Scopes;
6  using Platform.Data.Doublets.Memory.United.Specific;
7  using TLink = System.UInt64;
8
9  namespace Platform.Data.Doublets.Tests
10 {
11     public unsafe static class UnitedMemoryUInt64LinksTests
12     {
13         [Fact]
14         public static void CRUDTest()
15         {
16             Using(links => links.TestCRUDOperations());
17         }
18
19         [Fact]
20         public static void RawNumbersCRUDTest()

```

```

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

```

Index

./csharp/Platform.Data.Doublets.Tests/GenericLinksTests.cs, 157
./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs, 157
./csharp/Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs, 158
./csharp/Platform.Data.Doublets.Tests/ScopeTests.cs, 159
./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs, 159
./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt32LinksTests.cs, 160
./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt64LinksTests.cs, 161
./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs, 161
./csharp/Platform.Data.Doublets.Tests/UInt64LinksExtensionsTests.cs, 164
./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt32LinksTests.cs, 165
./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt64LinksTests.cs, 165
./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, 15
./csharp/Platform.Data.Doublets/ILinks.cs, 15
./csharp/Platform.Data.Doublets/ILinksExtensions.cs, 15
./csharp/Platform.Data.Doublets/ISynchronizedLinks.cs, 28
./csharp/Platform.Data.Doublets/Link.cs, 28
./csharp/Platform.Data.Doublets/LinkExtensions.cs, 31
./csharp/Platform.Data.Doublets/LinksOperatorBase.cs, 31
./csharp/Platform.Data.Doublets/Memory/ILinksListMethods.cs, 31
./csharp/Platform.Data.Doublets/Memory/ILinksTreeMethods.cs, 32
./csharp/Platform.Data.Doublets/Memory/IndexTreeType.cs, 32
./csharp/Platform.Data.Doublets/Memory/LinksHeader.cs, 32
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksRecursionlessSizeBalancedTreeMethodsBase.cs, 33
./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, 48
./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, 55
./csharp/Platform.Data.Doublets/Memory/Split/Generic/UnusedLinksListMethods.cs, 66
./csharp/Platform.Data.Doublets/Memory/Split/RawLinkDataPart.cs, 67
./csharp/Platform.Data.Doublets/Memory/Split/RawLinkIndexPart.cs, 67
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase.cs, 68
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSizeBalancedTreeMethodsBase.cs, 69
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 71
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSourcesSizeBalancedTreeMethods.cs, 72
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 73
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksTargetsSizeBalancedTreeMethods.cs, 74
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksRecursionlessSizeBalancedTreeMethodsBase.cs, 74

./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSizeBalancedTreeMethodsBase.cs, 76
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesLinkedListMethods.cs, 77
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 77
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesSizeBalancedTreeMethods.cs, 78
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 79
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksTargetsSizeBalancedTreeMethods.cs, 80
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32SplitMemoryLinks.cs, 81
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32UnusedLinksListMethods.cs, 83
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase.cs, 83
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSizeBalancedTreeMethodsBase.cs, 85
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 86
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSourcesSizeBalancedTreeMethods.cs, 87
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 88
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksTargetsSizeBalancedTreeMethods.cs, 89
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase.cs, 90
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSizeBalancedTreeMethodsBase.cs, 91
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesLinkedListMethods.cs, 92
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 92
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesSizeBalancedTreeMethods.cs, 93
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 94
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksTargetsSizeBalancedTreeMethods.cs, 95
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64SplitMemoryLinks.cs, 96
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64UnusedLinksListMethods.cs, 98
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksAvlBalancedTreeMethodsBase.cs, 98
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksRecursionlessSizeBalancedTreeMethodsBase.cs, 103
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSizeBalancedTreeMethodsBase.cs, 106
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesAvlBalancedTreeMethods.cs, 109
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 110
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesSizeBalancedTreeMethods.cs, 111
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsAvlBalancedTreeMethods.cs, 112
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 113
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsSizeBalancedTreeMethods.cs, 114
./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinks.cs, 115
./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinksBase.cs, 116
./csharp/Platform.Data.Doublets/Memory/United/Generic/UnusedLinksListMethods.cs, 123
./csharp/Platform.Data.Doublets/Memory/United/RawLink.cs, 124
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksRecursionlessSizeBalancedTreeMethodsBase.cs, 125
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSizeBalancedTreeMethodsBase.cs, 126
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 127
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSourcesSizeBalancedTreeMethods.cs, 128
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 129
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksTargetsSizeBalancedTreeMethods.cs, 130
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32UnitedMemoryLinks.cs, 131
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32UnusedLinksListMethods.cs, 133
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs, 133
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksRecursionlessSizeBalancedTreeMethodsBase.cs, 135
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs, 136
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs, 137
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 138
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.cs, 139
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs, 140
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 141
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs, 142
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnitedMemoryLinks.cs, 143
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnusedLinksListMethods.cs, 145
./csharp/Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 145
./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 146
./csharp/Platform.Data.Doublets/Stacks/Stack.cs, 147
./csharp/Platform.Data.Doublets/Stacks/StackExtensions.cs, 148
./csharp/Platform.Data.Doublets/SynchronizedLinks.cs, 148
./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs, 149
./csharp/Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 151