

LinksPlatform's Platform.Data.Doublets Class Library

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

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

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

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

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

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

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

```

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

```

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

```

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

```

```

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

```

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

```

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

```

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

```

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

```

```

5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Decorators
8 {
9     public class LinksItselfConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
10    {
11        private static readonly EqualityComparer<TLink> _equalityComparer =
12            ↪ EqualityComparer<TLink>.Default;
13
14        [MethodImpl(MethodImplOptions.AggressiveInlining)]
15        public LinksItselfConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
16
17        [MethodImpl(MethodImplOptions.AggressiveInlining)]
18        public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
19        {
20            var constants = _constants;
21            var itselfConstant = constants.Itself;
22            if (!_equalityComparer.Equals(constants.Any, itselfConstant) &&
23                ↪ restrictions.Contains(itselfConstant))
24            {
25                // Itself constant is not supported for Each method right now, skipping execution
26                return constants.Continue;
27            }
28            return _links.Each(handler, restrictions);
29        }
30
31        [MethodImpl(MethodImplOptions.AggressiveInlining)]
32        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
33            ↪ _links.Update(restrictions, _links.ResolveConstantAsSelfReference(_constants.Itself,
34            ↪ restrictions, substitution));
35    }
36 }

```

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

```

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

```

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

```

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

```

```

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

```

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

```

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

```

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

```

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

```

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

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

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

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

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

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

```

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

```

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

```

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

```

```

28 {
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public UInt64Links(ILinks<ulong> links) : base(links) { }
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     public override ulong Create(IList<ulong> restrictions) => _links.CreatePoint();
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
37     {
38         var constants = _constants;
39         var indexPartConstant = constants.IndexPart;
40         var sourcePartConstant = constants.SourcePart;
41         var targetPartConstant = constants.TargetPart;
42         var nullConstant = constants.Null;
43         var itselfConstant = constants.Itself;
44         var existedLink = nullConstant;
45         var updatedLink = restrictions[indexPartConstant];
46         var newSource = substitution[sourcePartConstant];
47         var newTarget = substitution[targetPartConstant];
48         var links = _links;
49         if (newSource != itselfConstant && newTarget != itselfConstant)
50         {
51             existedLink = links.SearchOrDefault(newSource, newTarget);
52         }
53         if (existedLink == nullConstant)
54         {
55             var before = links.GetLink(updatedLink);
56             if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
57                 ↪ newTarget)
58             {
59                 links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
60                     ↪ newSource,
61                                     newTarget == itselfConstant ? updatedLink :
62                     ↪ newTarget);
63             }
64             return updatedLink;
65         }
66         else
67         {
68             return _facade.MergeAndDelete(updatedLink, existedLink);
69         }
70     }
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     public override void Delete(IList<ulong> restrictions)
74     {
75         var linkIndex = restrictions[_constants.IndexPart];
76         var links = _links;
77         links.EnforceResetValues(linkIndex);
78         _facade.DeleteAllUsages(linkIndex);
79         links.Delete(linkIndex);
80     }
81 }
82
83 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using Platform.Collections;
5  using Platform.Collections.Lists;
6  using Platform.Data.Universal;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Decorators
11 {
12     /// <remarks>
13     /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
14     /// Now we go with nothing. And nothing is something one, but empty, and cannot be changed
15     ↪ by itself. But can cause creation (update from nothing) or deletion (update to nothing).
16     ///
17     /// TODO: Decide to change to IDoubletLinks or not to change. (Better to create
18     ↪ DefaultUniLinksBase, that contains logic itself and can be implemented using both
19     ↪ IDoubletLinks and ILinks.)
20     /// </remarks>
21     internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
22     {

```



```

20 private static readonly EqualityComparer<TLink> _equalityComparer =
    ↳ EqualityComparer<TLink>.Default;
21
22 public UniLinks(ILinks<TLink> links) : base(links) { }
23
24 private struct Transition
25 {
26     public IList<TLink> Before;
27     public IList<TLink> After;
28
29     public Transition(IList<TLink> before, IList<TLink> after)
30     {
31         Before = before;
32         After = after;
33     }
34 }
35
36 //public static readonly TLink NullConstant = Use<LinksConstants<TLink>>.Single.Null;
37 //public static readonly IReadOnlyList<TLink> NullLink = new
    ↳ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
    ↳ });
38
39 // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
    ↳ (Links-Expression)
40 public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
    ↳ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
    ↳ substitutedHandler)
41 {
42     /////List<Transition> transitions = null;
43     /////if (!restriction.IsNullOrEmpty())
44     /////{
45     /////    // Есть причина делать проход (чтение)
46     /////    if (matchedHandler != null)
47     /////    {
48     /////        if (!substitution.IsNullOrEmpty())
49     /////        {
50     /////            // restriction => { 0, 0, 0 } | { 0 } // Create
51     /////            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
    ↳ Create / Update
52     /////            // substitution => { 0, 0, 0 } | { 0 } // Delete
53     /////            transitions = new List<Transition>();
54     /////            if (Equals(substitution[Constants.IndexPart], Constants.Null))
55     /////            {
56     /////                // If index is Null, that means we always ignore every other
    ↳ value (they are also Null by definition)
57     /////                var matchDecision = matchedHandler(, NullLink);
58     /////                if (Equals(matchDecision, Constants.Break))
59     /////                {
60     /////                    return false;
61     /////                }
62     /////                if (!Equals(matchDecision, Constants.Skip))
63     /////                {
64     /////                    transitions.Add(new Transition(matchedLink, newValue));
65     /////                }
66     /////            }
67     /////            else
68     /////            {
69     /////                Func<T, bool> handler;
70     /////                handler = link =>
71     /////                {
72     /////                    var matchedLink = Memory.GetLinkValue(link);
73     /////                    var newValue = Memory.GetLinkValue(link);
74     /////                    newValue[Constants.IndexPart] = Constants.Itself;
75     /////                    newValue[Constants.SourcePart] =
    ↳ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
    ↳ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
76     /////                    newValue[Constants.TargetPart] =
    ↳ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
    ↳ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
77     /////                    var matchDecision = matchedHandler(matchedLink, newValue);
78     /////                    if (Equals(matchDecision, Constants.Break))
79     /////                    {
80     /////                        return false;
81     /////                    }
82     /////                    if (!Equals(matchDecision, Constants.Skip))
83     /////                    {
84     /////                        transitions.Add(new Transition(matchedLink, newValue));
85     /////                    }
86     /////                    return true;
87     /////                }
88     /////            }
89     /////        }
90     /////    }
91     /////    if (!Memory.Each(handler, restriction))
92     /////    {
93     /////        return Constants.Break;
94     /////    }
95     /////}
96 }

```

```

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

```

```

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

```

```

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

```

```

309     /// |         |         |-----| |
310     /// |         |         |   link   | |
311     /// |         |         |-----| |
312     /// |         |         |   change  | |
313     /// |-----|-----|
314     ///         changes
315     /// </remarks>
316     public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink>
    ↪ substitution)
317     {
318         var changes = new List<IList<IList<TLink>>>();
319         var @continue = _constants.Continue;
320         Trigger(condition, AlwaysContinue, substitution, (before, after) =>
321         {
322             var change = new[] { before, after };
323             changes.Add(change);
324             return @continue;
325         });
326         return changes;
327     }
328
329     private TLink AlwaysContinue(IList<TLink> linkToMatch) => _constants.Continue;
330 }
331 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets
8  {
9
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     /// <summury>
145     /// <para>.</para>
146     /// <para>.</para>
147     /// </summury>
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;

```

```

219 }
220
221 #endregion
222
223 /// <summary>
224 /// Возвращает индекс указанной связи.
225 /// </summary>
226 /// <param name="links">Хранилище связей.</param>
227 /// <param name="link">Связь представленная списком, состоящим из её адреса и
  ↳ содержимого.</param>
228 /// <returns>Индекс начальной связи для указанной связи.</returns>
229 [MethodImpl(MethodImplOptions.AggressiveInlining)]
230 public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
  ↳ link[links.Constants.IndexPart];
231
232 /// <summary>
233 /// Возвращает индекс начальной (Source) связи для указанной связи.
234 /// </summary>
235 /// <param name="links">Хранилище связей.</param>
236 /// <param name="link">Индекс связи.</param>
237 /// <returns>Индекс начальной связи для указанной связи.</returns>
238 [MethodImpl(MethodImplOptions.AggressiveInlining)]
239 public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
  ↳ links.GetLink(link)[links.Constants.SourcePart];
240
241 /// <summary>
242 /// Возвращает индекс начальной (Source) связи для указанной связи.
243 /// </summary>
244 /// <param name="links">Хранилище связей.</param>
245 /// <param name="link">Связь представленная списком, состоящим из её адреса и
  ↳ содержимого.</param>
246 /// <returns>Индекс начальной связи для указанной связи.</returns>
247 [MethodImpl(MethodImplOptions.AggressiveInlining)]
248 public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
  ↳ link[links.Constants.SourcePart];
249
250 /// <summary>
251 /// Возвращает индекс конечной (Target) связи для указанной связи.
252 /// </summary>
253 /// <param name="links">Хранилище связей.</param>
254 /// <param name="link">Индекс связи.</param>
255 /// <returns>Индекс конечной связи для указанной связи.</returns>
256 [MethodImpl(MethodImplOptions.AggressiveInlining)]
257 public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
  ↳ links.GetLink(link)[links.Constants.TargetPart];
258
259 /// <summary>
260 /// Возвращает индекс конечной (Target) связи для указанной связи.
261 /// </summary>
262 /// <param name="links">Хранилище связей.</param>
263 /// <param name="link">Связь представленная списком, состоящим из её адреса и
  ↳ содержимого.</param>
264 /// <returns>Индекс конечной связи для указанной связи.</returns>
265 [MethodImpl(MethodImplOptions.AggressiveInlining)]
266 public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
  ↳ link[links.Constants.TargetPart];
267
268 /// <summary>
269 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
  ↳ (handler) для каждой подходящей связи.
270 /// </summary>
271 /// <param name="links">Хранилище связей.</param>
272 /// <param name="handler">Обработчик каждой подходящей связи.</param>
273 /// <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>
274 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
  ↳ случае.</returns>
275 [MethodImpl(MethodImplOptions.AggressiveInlining)]
276 public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
  ↳ handler, params TLink[] restrictions)
277 => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
  ↳ links.Constants.Continue);
278
279 /// <summary>
280 /// 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)]
public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
    ↳ => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
    ↳ default) > 0;

348
349 #region Ensure
350 // TODO: May be move to EnsureExtensions or make it both there and here
351
352 [MethodImpl(MethodImplOptions.AggressiveInlining)]
353 public static void EnsureLinkExists<TLink>(this ILinks<TLink> links, IList<TLink>
    ↳ restrictions)
354 {
355     for (var i = 0; i < restrictions.Count; i++)
356     {
357         if (!links.Exists(restrictions[i]))
358         {
359             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
    ↳ $"sequence[{i}]");
360         }
361     }
362 }
363
364 [MethodImpl(MethodImplOptions.AggressiveInlining)]
365 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
    ↳ reference, string argumentName)
366 {
367     if (links.Constants.IsInternalReference(reference) && !links.Exists(reference))
368     {
369         throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
370     }
371 }
372
373 [MethodImpl(MethodImplOptions.AggressiveInlining)]
374 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
    ↳ IList<TLink> restrictions, string argumentName)
375 {
376     for (int i = 0; i < restrictions.Count; i++)
377     {
378         links.EnsureInnerReferenceExists(restrictions[i], argumentName);
379     }
380 }
381
382 [MethodImpl(MethodImplOptions.AggressiveInlining)]
383 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
    ↳ restrictions)
384 {
385     var equalityComparer = EqualityComparer<TLink>.Default;
386     var any = links.Constants.Any;
387     for (var i = 0; i < restrictions.Count; i++)
388     {
389         if (!equalityComparer.Equals(restrictions[i], any) &&
    ↳ !links.Exists(restrictions[i]))
390         {
391             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
    ↳ $"sequence[{i}]");
392         }
393     }
394 }
395
396 [MethodImpl(MethodImplOptions.AggressiveInlining)]
397 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
    ↳ string argumentName)
398 {
399     var equalityComparer = EqualityComparer<TLink>.Default;
400     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
401     {
402         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
403     }

```

```

404 }
405
406 [MethodImpl(MethodImplOptions.AggressiveInlining)]
407 public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
    ↳ link, string argumentName)
408 {
409     var equalityComparer = EqualityComparer<TLink>.Default;
410     if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
411     {
412         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
413     }
414 }
415
416 /// <param name="links">Хранилище связей.</param>
417 [MethodImpl(MethodImplOptions.AggressiveInlining)]
418 public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
    ↳ TLink target)
419 {
420     if (links.Exists(source, target))
421     {
422         throw new LinkWithSameValueAlreadyExistsException();
423     }
424 }
425
426 /// <param name="links">Хранилище связей.</param>
427 [MethodImpl(MethodImplOptions.AggressiveInlining)]
428 public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
429 {
430     if (links.HasUsages(link))
431     {
432         throw new ArgumentLinkHasDependenciesException<TLink>(link);
433     }
434 }
435
436 /// <param name="links">Хранилище связей.</param>
437 [MethodImpl(MethodImplOptions.AggressiveInlining)]
438 public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ addresses) => links.EnsureCreated(links.Create, addresses);
439
440 /// <param name="links">Хранилище связей.</param>
441 [MethodImpl(MethodImplOptions.AggressiveInlining)]
442 public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
443
444 /// <param name="links">Хранилище связей.</param>
445 [MethodImpl(MethodImplOptions.AggressiveInlining)]
446 public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
    ↳ params TLink[] addresses)
447 {
448     var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
449     var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
450     var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
        ↳ !links.Exists(x)));
451     if (nonExistentAddresses.Count > 0)
452     {
453         var max = nonExistentAddresses.Max();
454         max = uInt64ToAddressConverter.Convert(System.Math.Min(addressToUInt64Converter.
            ↳ Convert(max),
            ↳ addressToUInt64Converter.Convert(links.Constants.InternalReferencesRange.Max
            ↳ imum)));
455         var createdLinks = new List<TLink>();
456         var equalityComparer = EqualityComparer<TLink>.Default;
457         TLink createdLink = creator();
458         while (!equalityComparer.Equals(createdLink, max))
459         {
460             createdLinks.Add(createdLink);
461         }
462         for (var i = 0; i < createdLinks.Count; i++)
463         {
464             if (!nonExistentAddresses.Contains(createdLinks[i]))
465             {
466                 links.Delete(createdLinks[i]);
467             }
468         }
469     }
470 }
471
472 #endregion

```

```

473
474 /// <param name="links">Хранилище связей.</param>
475 [MethodImpl(MethodImplOptions.AggressiveInlining)]
476 public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
477 {
478     var constants = links.Constants;
479     var values = links.GetLink(link);
480     TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link,
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/Incrementers/FrequencyIncrementer.cs

```

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

```

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

```

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

```

```

17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     public TLink Increment(TLink unaryNumber)
19     {
20         var links = _links;
21         if (_equalityComparer.Equals(unaryNumber, _unaryOne))
22         {
23             return links.GetOrCreate(_unaryOne, _unaryOne);
24         }
25         var source = links.GetSource(unaryNumber);
26         var target = links.GetTarget(unaryNumber);
27         if (_equalityComparer.Equals(source, target))
28         {
29             return links.GetOrCreate(unaryNumber, _unaryOne);
30         }
31         else
32         {
33             return links.GetOrCreate(source, Increment(target));
34         }
35     }
36 }
37 }
38 }

```

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

```

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

```

```

54 public Link(ref Link<TLink> other) => SetValues(ref other, out Index, out Source, out
55     ↪ Target);
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 public Link(TLink index, TLink source, TLink target)
59 {
60     Index = index;
61     Source = source;
62     Target = target;
63 }
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 private static void SetValues(ref Link<TLink> other, out TLink index, out TLink source,
67     ↪ out TLink target)
68 {
69     index = other.Index;
70     source = other.Source;
71     target = other.Target;
72 }
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 private static void SetValues(ICollection<TLink> values, out TLink index, out TLink source,
76     ↪ out TLink target)
77 {
78     switch (values.Count)
79     {
80         case 3:
81             index = values[0];
82             source = values[1];
83             target = values[2];
84             break;
85         case 2:
86             index = values[0];
87             source = values[1];
88             target = default;
89             break;
90         case 1:
91             index = values[0];
92             source = default;
93             target = default;
94             break;
95         default:
96             index = default;
97             source = default;
98             target = default;
99             break;
100     }
101 }
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 public override int GetHashCode() => (Index, Source, Target).GetHashCode();
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
108     && _equalityComparer.Equals(Source, _constants.Null)
109     && _equalityComparer.Equals(Target, _constants.Null);
110
111 [MethodImpl(MethodImplOptions.AggressiveInlining)]
112 public override bool Equals(object other) => other is Link<TLink> &&
113     ↪ Equals((Link<TLink>)other);
114
115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
117     && _equalityComparer.Equals(Source, other.Source)
118     && _equalityComparer.Equals(Target, other.Target);
119
120 [MethodImpl(MethodImplOptions.AggressiveInlining)]
121 public static string ToString(TLink index, TLink source, TLink target) => $"{index}:
122     ↪ {source}->{target}";
123
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 public static string ToString(TLink source, TLink target) => $"{source}->{target}";
126
127 [MethodImpl(MethodImplOptions.AggressiveInlining)]
128 public static implicit operator TLink[] (Link<TLink> link) => link.ToArray();
129
130 [MethodImpl(MethodImplOptions.AggressiveInlining)]
131 public static implicit operator Link<TLink> (TLink[] linkArray) => new
132     ↪ Link<TLink>(linkArray);

```

```

128 [MethodImpl(MethodImplOptions.AggressiveInlining)]
129 public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
    ↳ ToString(Source, Target) : ToString(Index, Source, Target);
130
131 #region IList
132
133 public int Count
134 {
135     [MethodImpl(MethodImplOptions.AggressiveInlining)]
136     get => Length;
137 }
138
139 public bool IsReadOnly
140 {
141     [MethodImpl(MethodImplOptions.AggressiveInlining)]
142     get => true;
143 }
144
145 public TLink this[int index]
146 {
147     [MethodImpl(MethodImplOptions.AggressiveInlining)]
148     get
149     {
150         Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
151             ↳ nameof(index));
152         if (index == _constants.IndexPart)
153         {
154             return Index;
155         }
156         if (index == _constants.SourcePart)
157         {
158             return Source;
159         }
160         if (index == _constants.TargetPart)
161         {
162             return Target;
163         }
164         throw new NotSupportedException(); // Impossible path due to
165             ↳ Ensure.ArgumentInRange
166     }
167     [MethodImpl(MethodImplOptions.AggressiveInlining)]
168     set => throw new NotSupportedException();
169 }
170
171 [MethodImpl(MethodImplOptions.AggressiveInlining)]
172 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
173
174 [MethodImpl(MethodImplOptions.AggressiveInlining)]
175 public IEnumerator<TLink> GetEnumerator()
176 {
177     yield return Index;
178     yield return Source;
179     yield return Target;
180 }
181
182 [MethodImpl(MethodImplOptions.AggressiveInlining)]
183 public void Add(TLink item) => throw new NotSupportedException();
184
185 [MethodImpl(MethodImplOptions.AggressiveInlining)]
186 public void Clear() => throw new NotSupportedException();
187
188 [MethodImpl(MethodImplOptions.AggressiveInlining)]
189 public bool Contains(TLink item) => IndexOf(item) >= 0;
190
191 [MethodImpl(MethodImplOptions.AggressiveInlining)]
192 public void CopyTo(TLink[] array, int arrayIndex)
193 {
194     Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
195     Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
196         ↳ nameof(arrayIndex));
197     if (arrayIndex + Length > array.Length)
198     {
199         throw new InvalidOperationException();
200     }
201     array[arrayIndex++] = Index;
202     array[arrayIndex++] = Source;
203     array[arrayIndex] = Target;
204 }

```

```

203 [MethodImpl(MethodImplOptions.AggressiveInlining)]
204 public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
205
206 [MethodImpl(MethodImplOptions.AggressiveInlining)]
207 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) =>
    ↪ left.Equals(right);
232
233 [MethodImpl(MethodImplOptions.AggressiveInlining)]
234 public static bool operator !=(Link<TLink> left, Link<TLink> right) => !(left == right);
235
236 #endregion
237 }
238 }

```

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

```

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

```

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

```

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

```


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

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

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

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

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

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

1.30 ./csharp/Platform.Data.Doublets/Memory/LinksHeader.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Unsafe;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Memory
9 {
10    public struct LinksHeader<TLink> : IEquatable<LinksHeader<TLink>>
11    {
12        private static readonly EqualityComparer<TLink> _equalityComparer =
13            ↳ EqualityComparer<TLink>.Default;
14
15        public static readonly long SizeInBytes = Structure<LinksHeader<TLink>>.Size;
16
17        public TLink AllocatedLinks;
18        public TLink ReservedLinks;
19        public TLink FreeLinks;
20        public TLink FirstFreeLink;
21    }
22 }
```

```

20     public TLink RootAsSource;
21     public TLink RootAsTarget;
22     public TLink LastFreeLink;
23     public TLink Reserved8;
24
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 }

```

1.31 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksRecursionlessSizeBalancedTreeMethods

```

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> :
        ↳ RecursionlessSizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
14     {
15         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            ↳ UncheckedConverter<TLink, long>.Default;
16
17         protected readonly TLink Break;
18         protected readonly TLink Continue;
19         protected readonly byte* LinksDataParts;
20         protected readonly byte* LinksIndexParts;
21         protected readonly byte* Header;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
            ↳ 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();
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected abstract TLink GetBasePartValue(TLink link);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
            ↳ rootSource, TLink rootTarget);
41

```

```

42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected abstract bool FirstIsToLeftOfSecond(TLink source, TLink target, TLink
    ↪ rootSource, TLink rootTarget);
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
    ↪ AsRef<LinksHeader<TLink>>(Header);
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
    ↪ AsRef<RawLinkDataPart<TLink>>(LinksDataParts + (RawLinkDataPart<TLink>.SizeInBytes *
    ↪ _addressToInt64Converter.Convert(link)));
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
    ↪ ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
    ↪ (RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link)));
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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected override bool FirstIsToLeftOfSecond(TLink first, TLink second)
63 {
64     ref var firstLink = ref GetLinkDataPartReference(first);
65     ref var secondLink = ref GetLinkDataPartReference(second);
66     return FirstIsToLeftOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
67 }
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
71 {
72     ref var firstLink = ref GetLinkDataPartReference(first);
73     ref var secondLink = ref GetLinkDataPartReference(second);
74     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
75 }
76
77 public TLink this[TLink index]
78 {
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     get
81     {
82         var root = GetTreeRoot();
83         if (GreaterOrEqualThan(index, GetSize(root)))
84         {
85             return Zero;
86         }
87         while (!EqualToZero(root))
88         {
89             var left = GetLeftOrDefault(root);
90             var leftSize = GetSizeOrZero(left);
91             if (LessThan(index, leftSize))
92             {
93                 root = left;
94                 continue;
95             }
96             if (AreEqual(index, leftSize))
97             {
98                 return root;
99             }
100             root = GetRightOrDefault(root);
101             index = Subtract(index, Increment(leftSize));
102         }
103         return Zero; // TODO: Impossible situation exception (only if tree structure
    ↪ broken)
104     }
105 }
106
107 /// <summary>
108 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
    ↪ (концом).
109 /// </summary>

```

```

110 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
111 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
112 /// <returns>Индекс искомой связи.</returns>
113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
114 public TLink Search(TLink source, TLink target)
115 {
116     var root = GetTreeRoot();
117     while (!EqualToZero(root))
118     {
119         ref var rootLink = ref GetLinkDataPartReference(root);
120         var rootSource = rootLink.Source;
121         var rootTarget = rootLink.Target;
122         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
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;

```

```

184         if (EqualToZero(link))
185         {
186             return @continue;
187         }
188         var linkBasePart = GetBasePartValue(link);
189         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 }

```

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

```

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

```

```

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();
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 protected abstract TLink GetBasePartValue(TLink link);
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
    ↪ rootSource, TLink rootTarget);
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
    ↪ rootSource, TLink rootTarget);
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
    ↪ AsRef<LinksHeader<TLink>>(Header);
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
    ↪ AsRef<RawLinkDataPart<TLink>>(LinksDataParts + (RawLinkDataPart<TLink>.SizeInBytes *
    ↪ _addressToInt64Converter.Convert(link)));
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
    ↪ ref AsRef<RawLinkIndexPart<TLink>>(LinksIndexParts +
    ↪ (RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link)));
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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
63 {
64     ref var firstLink = ref GetLinkDataPartReference(first);
65     ref var secondLink = ref GetLinkDataPartReference(second);
66     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
67 }
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
71 {
72     ref var firstLink = ref GetLinkDataPartReference(first);
73     ref var secondLink = ref GetLinkDataPartReference(second);
74     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
75 }
76
77 public TLink this[TLink index]
78 {
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     get
81     {
82         var root = GetTreeRoot();
83         if (GreaterOrEqualThan(index, GetSize(root)))
84         {
85             return Zero;
86         }
87         while (!EqualToZero(root))
88         {
89             var left = GetLeftOrDefault(root);
90             var leftSize = GetSizeOrZero(left);
91             if (LessThan(index, leftSize))
92             {
93                 root = left;
94                 continue;
95             }

```

```

96         if (AreEqual(index, leftSize))
97         {
98             return root;
99         }
100         root = GetRightOrDefault(root);
101         index = Subtract(index, Increment(leftSize));
102     }
103     return Zero; // TODO: Impossible situation exception (only if tree structure
104     ↪ broken)
105 }
106
107 /// <summary>
108 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
109 ↪ (концом).
110 /// </summary>
111 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
112 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
113 /// <returns>Индекс искомой связи.</returns>
114 [MethodImpl(MethodImplOptions.AggressiveInlining)]
115 public TLink Search(TLink source, TLink target)
116 {
117     var root = GetTreeRoot();
118     while (!EqualToZero(root))
119     {
120         ref var rootLink = ref GetLinkDataPartReference(root);
121         var rootSource = rootLink.Source;
122         var rootTarget = rootLink.Target;
123         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
124             ↪ node.Key < root.Key
125         {
126             root = GetLeftOrDefault(root);
127         }
128         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
129             ↪ node.Key > root.Key
130         {
131             root = GetRightOrDefault(root);
132         }
133         else // node.Key == root.Key
134         {
135             return root;
136         }
137     }
138     return Zero;
139 }
140
141 // TODO: Return indices range instead of references count
142 [MethodImpl(MethodImplOptions.AggressiveInlining)]
143 public TLink CountUsages(TLink link)
144 {
145     var root = GetTreeRoot();
146     var total = GetSize(root);
147     var totalRightIgnore = Zero;
148     while (!EqualToZero(root))
149     {
150         var @base = GetBasePartValue(root);
151         if (LessOrEqualThan(@base, link))
152         {
153             root = GetRightOrDefault(root);
154         }
155         else
156         {
157             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
158             root = GetLeftOrDefault(root);
159         }
160     }
161     root = GetTreeRoot();
162     var totalLeftIgnore = Zero;
163     while (!EqualToZero(root))
164     {
165         var @base = GetBasePartValue(root);
166         if (GreaterOrEqualThan(@base, link))
167         {
168             root = GetLeftOrDefault(root);
169         }
170         else
171         {
172             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
173         }
174     }
175 }

```

```

170         root = GetRightOrDefault(root);
171     }
172 }
173 return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
174 }
175
176 [MethodImpl(MethodImplOptions.AggressiveInlining)]
177 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
178     ↳ EachUsageCore(@base, GetTreeRoot(), handler);
179
180 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
181 ↳ low-level MSIL stack.
182 [MethodImpl(MethodImplOptions.AggressiveInlining)]
183 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
184 {
185     var @continue = Continue;
186     if (EqualToZero(link))
187     {
188         return @continue;
189     }
190     var linkBasePart = GetBasePartValue(link);
191     var @break = Break;
192     if (GreaterThan(linkBasePart, @base))
193     {
194         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
195         {
196             return @break;
197         }
198     }
199     else if (LessThan(linkBasePart, @base))
200     {
201         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
202         {
203             return @break;
204         }
205     }
206     else //if (linkBasePart == @base)
207     {
208         if (AreEqual(handler(GetLinkValues(link)), @break))
209         {
210             return @break;
211         }
212         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
213         {
214             return @break;
215         }
216         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
217         {
218             return @break;
219         }
220     }
221     return @continue;
222 }
223
224 [MethodImpl(MethodImplOptions.AggressiveInlining)]
225 protected override void PrintNodeValue(TLink node, StringBuilder sb)
226 {
227     ref var link = ref GetLinkDataPartReference(node);
228     sb.Append(' ');
229     sb.Append(link.Source);
230     sb.Append('-');
231     sb.Append('>');
232     sb.Append(link.Target);
233 }

```

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

```

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

```



```

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

```

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

```

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

```

```

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

```

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

```

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

```

```

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

```

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

```

1     using System.Runtime.CompilerServices;
2
3     #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5     namespace Platform.Data.Doublets.Memory.Split.Generic
6     {
7         public unsafe class ExternalLinksTargetsSizeBalancedTreeMethods<TLink> :
8             ↳ ExternalLinksSizeBalancedTreeMethodsBase<TLink>
9         {
10             [MethodImpl(MethodImplOptions.AggressiveInlining)]
11             public ExternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants,
12                 ↳ byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
13                 ↳ linksDataParts, linksIndexParts, header) { }
14
15             [MethodImpl(MethodImplOptions.AggressiveInlining)]
16             protected override ref TLink GetLeftReference(TLink node) => ref
17                 ↳ GetLinkIndexPartReference(node).LeftAsTarget;
18
19
20

```

```

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

```

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

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7  using static System.Runtime.CompilerServices.Unsafe;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Memory.Split.Generic
12 {
13     public unsafe abstract class InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink> :
    ↪ RecursionlessSizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
14     {
15         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
    ↪ UncheckedConverter<TLink, long>.Default;
16
17         protected readonly TLink Break;
18         protected readonly TLink Continue;
19         protected readonly byte* LinksDataParts;
20         protected readonly byte* LinksIndexParts;

```

```

21     protected readonly byte* Header;
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     protected InternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
    ↪ 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 FirstIsToLeftOfSecond(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
93     ↪ (концом).
94     /// </summary>
95     /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
96     /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
97     /// <returns>Индекс искомой связи.</returns>
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     public abstract TLink Search(TLink source, TLink target);
100
101     [MethodImpl(MethodImplOptions.AggressiveInlining)]
102     protected TLink SearchCore(TLink root, TLink key)
103     {
104         while (!EqualToZero(root))
105         {
106             var rootKey = GetKeyPartValue(root);
107             if (LessThan(key, rootKey)) // node.Key < root.Key
108             {
109                 root = GetLeftOrDefault(root);
110             }
111             else if (GreaterThan(key, rootKey)) // node.Key > root.Key
112             {
113                 root = GetRightOrDefault(root);
114             }
115             else // node.Key == root.Key
116             {
117                 return root;
118             }
119         }
120         return Zero;
121     }
122
123     /// TODO: Return indices range instead of references count
124     [MethodImpl(MethodImplOptions.AggressiveInlining)]
125     public TLink CountUsages(TLink link) => GetSizeOrZero(GetTreeRoot(link));
126
127     [MethodImpl(MethodImplOptions.AggressiveInlining)]
128     public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
129     ↪ EachUsageCore(@base, GetTreeRoot(@base), handler);
130
131     /// TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
132     ↪ low-level MSIL stack.
133     [MethodImpl(MethodImplOptions.AggressiveInlining)]
134     private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
135     {
136         var @continue = Continue;
137         if (EqualToZero(link))
138         {
139             return @continue;
140         }
141         var @break = Break;
142         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
143         {
144             return @break;
145         }
146         if (AreEqual(handler(GetLinkValues(link)), @break))
147         {
148             return @break;
149         }
150         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
151         {
152             return @break;
153         }
154         return @continue;
155     }
156
157     [MethodImpl(MethodImplOptions.AggressiveInlining)]
158     protected override void PrintNodeValue(TLink node, StringBuilder sb)
159     {
160         ref var link = ref GetLinkDataPartReference(node);
161         sb.Append(' ');
162         sb.Append(link.Source);
163         sb.Append('-');
164         sb.Append('>');
165         sb.Append(link.Target);
166     }
167 }

```

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

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

```

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

```



```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Methods.Lists;
5  using Platform.Converters;
6  using static System.Runtime.CompilerServices.Unsafe;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Memory.Split.Generic
11 {
12     public unsafe class InternalLinksSourcesLinkedListMethods<TLink> :
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             [MethodImpl(MethodImplOptions.AggressiveInlining)]
32             protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
33                 ↳ AsRef<RawLinkDataPart<TLink>>(_linksDataParts + (RawLinkDataPart<TLink>.SizeInBytes
34                 ↳ * _addressToInt64Converter.Convert(link)));
35
36             [MethodImpl(MethodImplOptions.AggressiveInlining)]
37             protected virtual ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
38                 ↳ ref AsRef<RawLinkIndexPart<TLink>>(_linksIndexParts +
39                 ↳ (RawLinkIndexPart<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link)));
40
41             [MethodImpl(MethodImplOptions.AggressiveInlining)]
42             protected override TLink GetFirst(TLink head) =>
43                 ↳ GetLinkIndexPartReference(head).RootAsSource;
44
45             [MethodImpl(MethodImplOptions.AggressiveInlining)]
46             protected override TLink GetLast(TLink head)
47             {
48                 var first = GetLinkIndexPartReference(head).RootAsSource;
49                 if (EqualToZero(first))
50                 {
51                     return first;
52                 }
53                 else
54                 {
55                     return GetPrevious(first);
56                 }
57             }
58         }
59     }
60 }

```

```

51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override TLink GetPrevious(TLink element) =>
53     ↪ GetLinkIndexPartReference(element).LeftAsSource;
54
55 [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 }
125 }

```

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

```

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

```

1.41 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesSizeBalancedTreeMethods.cs

```

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

```

```

5 namespace Platform.Data.Doublets.Memory.Split.Generic
6 {
7     public unsafe class InternalLinksSourcesSizeBalancedTreeMethods<TLink> :
8         ↳ InternalLinksSizeBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants,
12             ↳ byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
13             ↳ linksDataParts, linksIndexParts, header) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected override ref TLink GetLeftReference(TLink node) => ref
17             ↳ GetLinkIndexPartReference(node).LeftAsSource;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected override ref TLink GetRightReference(TLink node) => ref
21             ↳ GetLinkIndexPartReference(node).RightAsSource;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override TLink GetLeft(TLink node) =>
25             ↳ GetLinkIndexPartReference(node).LeftAsSource;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override TLink GetRight(TLink node) =>
29             ↳ GetLinkIndexPartReference(node).RightAsSource;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetLeft(TLink node, TLink left) =>
33             ↳ GetLinkIndexPartReference(node).LeftAsSource = left;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override void SetRight(TLink node, TLink right) =>
37             ↳ GetLinkIndexPartReference(node).RightAsSource = right;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override TLink GetSize(TLink node) =>
41             ↳ GetLinkIndexPartReference(node).SizeAsSource;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override void SetSize(TLink node, TLink size) =>
45             ↳ GetLinkIndexPartReference(node).SizeAsSource = size;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override TLink GetTreeRoot(TLink link) =>
49             ↳ GetLinkIndexPartReference(link).RootAsSource;
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override TLink GetBasePartValue(TLink link) =>
53             ↳ GetLinkDataPartReference(link).Source;
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override TLink GetKeyPartValue(TLink link) =>
57             ↳ GetLinkDataPartReference(link).Target;
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override void ClearNode(TLink node)
61         {
62             ref var link = ref GetLinkIndexPartReference(node);
63             link.LeftAsSource = Zero;
64             link.RightAsSource = Zero;
65             link.SizeAsSource = Zero;
66         }
67
68         public override TLink Search(TLink source, TLink target) =>
69             ↳ SearchCore(GetTreeRoot(source), target);
70     }
71 }

```

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

```

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

```

```

9      [MethodImpl(MethodImplOptions.AggressiveInlining)]
10     public InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(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
18     [MethodImpl(MethodImplOptions.AggressiveInlining)]
19     protected override TLink GetLeft(TLink node) =>
    ↳ GetLinkIndexPartReference(node).LeftAsTarget;

20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     protected override TLink GetRight(TLink node) =>
    ↳ GetLinkIndexPartReference(node).RightAsTarget;

23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     protected override void SetLeft(TLink node, TLink left) =>
    ↳ GetLinkIndexPartReference(node).LeftAsTarget = left;

26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     protected override void SetRight(TLink node, TLink right) =>
    ↳ GetLinkIndexPartReference(node).RightAsTarget = right;

29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     protected override TLink GetSize(TLink node) =>
    ↳ GetLinkIndexPartReference(node).SizeAsTarget;

32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override void SetSize(TLink node, TLink size) =>
    ↳ GetLinkIndexPartReference(node).SizeAsTarget = size;

35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected override TLink GetTreeRoot(TLink link) =>
    ↳ GetLinkIndexPartReference(link).RootAsTarget;

38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected override TLink GetBasePartValue(TLink link) =>
    ↳ GetLinkDataPartReference(link).Target;

41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override TLink GetKeyPartValue(TLink link) =>
    ↳ GetLinkDataPartReference(link).Source;

44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override void ClearNode(TLink node)
47     {
48         ref var link = ref GetLinkIndexPartReference(node);
49         link.LeftAsTarget = Zero;
50         link.RightAsTarget = Zero;
51         link.SizeAsTarget = Zero;
52     }

53
54     public override TLink Search(TLink source, TLink target) =>
    ↳ SearchCore(GetTreeRoot(target), source);

55 }
56 }

```

1.43 ./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsSizeBalancedTreeMethods.cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.Split.Generic
6  {
7      public unsafe class InternalLinksTargetsSizeBalancedTreeMethods<TLink> :
    ↳ InternalLinksSizeBalancedTreeMethodsBase<TLink>
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public InternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants,
    ↳ byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
    ↳ linksDataParts, linksIndexParts, header) { }

```

```

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

```

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

```

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

```

```

21 [MethodImpl(MethodImplOptions.AggressiveInlining)]
22 public SplitMemoryLinks(string dataMemory, string indexMemory) : this(new
    ↳ FileMappedResizableDirectMemory(dataMemory), new
    ↳ FileMappedResizableDirectMemory(indexMemory)) { }
23
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
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     protected override void ResetPointers()
75     {
76         base.ResetPointers();
77         _linksDataParts = null;
78         _linksIndexParts = null;
79         _header = null;
80     }
81
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     protected override ref LinksHeader<TLink> GetHeaderReference() => ref
84         ↪ AsRef<LinksHeader<TLink>>(_header);
85
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
88         ↪ => ref AsRef<RawLinkDataPart<TLink>>(_linksDataParts + (LinkDataPartSizeInBytes *
89         ↪ ConvertToInt64(linkIndex)));
90
91     [MethodImpl(MethodImplOptions.AggressiveInlining)]
92     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
93         ↪ linkIndex) => ref AsRef<RawLinkIndexPart<TLink>>(_linksIndexParts +
94         ↪ (LinkIndexPartSizeInBytes * ConvertToInt64(linkIndex)));
95 }
96 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5  using Platform.Singletons;
6  using Platform.Converters;
7  using Platform.Numbers;
8  using Platform.Memory;
9  using Platform.Data.Exceptions;
10
11  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13  namespace Platform.Data.Doublets.Memory.Split.Generic
14  {
15      public abstract class SplitMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16      {
17          private static readonly EqualityComparer<TLink> _equalityComparer =
18              ↪ EqualityComparer<TLink>.Default;
19          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
20          private static readonly uncheckedConverter<TLink, long> _addressToInt64Converter =
21              ↪ uncheckedConverter<TLink, long>.Default;
22          private static readonly uncheckedConverter<long, TLink> _int64ToAddressConverter =
23              ↪ uncheckedConverter<long, TLink>.Default;
24
25          private static readonly TLink _zero = default;
26          private static readonly TLink _one = Arithmetic.Increment(_zero);
27
28          /// <summary>Возвращает размер одной связи в байтах.</summary>
29          /// <remarks>
30          /// Используется только во вне класса, не рекомендуется использовать внутри.
31          /// Так как во вне не обязательно будет доступен unsafe C#.
32          /// </remarks>
33          public static readonly long LinkDataPartSizeInBytes = RawLinkDataPart<TLink>.SizeInBytes;
34
35          public static readonly long LinkIndexPartSizeInBytes =
36              ↪ RawLinkIndexPart<TLink>.SizeInBytes;
37
38          public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
39
40          public static readonly long DefaultLinksSizeStep = 1 * 1024 * 1024;
41
42          protected readonly IResizableDirectMemory _dataMemory;
43          protected readonly IResizableDirectMemory _indexMemory;
44          protected readonly bool _useLinkedList;
45          protected readonly long _dataMemoryReservationStepInBytes;
46          protected readonly long _indexMemoryReservationStepInBytes;
47
48          protected InternalLinksSourcesLinkedListMethods<TLink> InternalSourcesListMethods;
49          protected ILinksTreeMethods<TLink> InternalSourcesTreeMethods;

```



```

46 protected ILinksTreeMethods<TLink> ExternalSourcesTreeMethods;
47 protected ILinksTreeMethods<TLink> InternalTargetsTreeMethods;
48 protected ILinksTreeMethods<TLink> ExternalTargetsTreeMethods;
49 // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
    ↳ нужно использовать не список а дерево, так как так можно быстрее проверить на
    ↳ наличие связи внутри
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
    ↳ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants, bool
    ↳ useLinkedList)
73 {
74     _dataMemory = dataMemory;
75     _indexMemory = indexMemory;
76     _dataMemoryReservationStepInBytes = memoryReservationStep * LinkDataPartSizeInBytes;
77     _indexMemoryReservationStepInBytes = memoryReservationStep *
    ↳ LinkIndexPartSizeInBytes;
78     _useLinkedList = useLinkedList;
79     Constants = constants;
80 }
81
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
    ↳ memoryReservationStep, Default<LinksConstants<TLink>>.Instance, useLinkedList: true)
    ↳ { }
84
85 [MethodImpl(MethodImplOptions.AggressiveInlining)]
86 protected virtual void Init(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory)
87 {
88     // Read allocated links from header
89     if (indexMemory.ReservedCapacity < LinkHeaderSizeInBytes)
90     {
91         indexMemory.ReservedCapacity = LinkHeaderSizeInBytes;
92     }
93     SetPointers(dataMemory, indexMemory);
94     ref var header = ref GetHeaderReference();
95     var allocatedLinks = ConvertToInt64(header.AllocatedLinks);
96     // Adjust reserved capacity
97     var minimumDataReservedCapacity = allocatedLinks * LinkDataPartSizeInBytes;
98     if (minimumDataReservedCapacity < dataMemory.UsedCapacity)
99     {
100         minimumDataReservedCapacity = dataMemory.UsedCapacity;
101     }
102     if (minimumDataReservedCapacity < _dataMemoryReservationStepInBytes)
103     {
104         minimumDataReservedCapacity = _dataMemoryReservationStepInBytes;
105     }
106     var minimumIndexReservedCapacity = allocatedLinks * LinkDataPartSizeInBytes;
107     if (minimumIndexReservedCapacity < indexMemory.UsedCapacity)
108     {
109         minimumIndexReservedCapacity = indexMemory.UsedCapacity;
110     }
111     if (minimumIndexReservedCapacity < _indexMemoryReservationStepInBytes)
112     {
113         minimumIndexReservedCapacity = _indexMemoryReservationStepInBytes;
114     }
115     // Check for alignment

```

```

116     if (minimumDataReservedCapacity % _dataMemoryReservationStepInBytes > 0)
117     {
118         minimumDataReservedCapacity = ((minimumDataReservedCapacity /
119             ↪ _dataMemoryReservationStepInBytes) * _dataMemoryReservationStepInBytes) +
120             ↪ _dataMemoryReservationStepInBytes;
121     }
122     if (minimumIndexReservedCapacity % _indexMemoryReservationStepInBytes > 0)
123     {
124         minimumIndexReservedCapacity = ((minimumIndexReservedCapacity /
125             ↪ _indexMemoryReservationStepInBytes) * _indexMemoryReservationStepInBytes) +
126             ↪ _indexMemoryReservationStepInBytes;
127     }
128     if (dataMemory.ReservedCapacity != minimumDataReservedCapacity)
129     {
130         dataMemory.ReservedCapacity = minimumDataReservedCapacity;
131     }
132     if (indexMemory.ReservedCapacity != minimumIndexReservedCapacity)
133     {
134         indexMemory.ReservedCapacity = minimumIndexReservedCapacity;
135     }
136     SetPointers(dataMemory, indexMemory);
137     header = ref GetHeaderReference();
138     // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
139     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
140     dataMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
141         ↪ LinkDataPartSizeInBytes) + LinkDataPartSizeInBytes; // First link is read only
142     ↪ zero link.
143     indexMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
144         ↪ LinkIndexPartSizeInBytes) + LinkHeaderSizeInBytes;
145     // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
146     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
147     header.ReservedLinks = ConvertToAddress((dataMemory.ReservedCapacity -
148         ↪ LinkDataPartSizeInBytes) / LinkDataPartSizeInBytes);
149 }
150
151 [MethodImpl(MethodImplOptions.AggressiveInlining)]
152 public virtual TLink Count(IList<TLink> restrictions)
153 {
154     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
155     if (restrictions.Count == 0)
156     {
157         return Total;
158     }
159     var constants = Constants;
160     var any = constants.Any;
161     var index = restrictions[constants.IndexPart];
162     if (restrictions.Count == 1)
163     {
164         if (AreEqual(index, any))
165         {
166             return Total;
167         }
168         return Exists(index) ? GetOne() : GetZero();
169     }
170     if (restrictions.Count == 2)
171     {
172         var value = restrictions[1];
173         if (AreEqual(index, 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                 // ...
194             }
195         }
196     }
197 }

```

```

183         {
184             return Add(InternalSourcesTreeMethods.CountUsages(value),
185                 ↪ InternalTargetsTreeMethods.CountUsages(value));
186         }
187     }
188 }
189 else
190 {
191     if (!Exists(index))
192     {
193         return GetZero();
194     }
195     if (AreEqual(value, any))
196     {
197         return GetOne();
198     }
199     ref var storedLinkValue = ref GetLinkDataPartReference(index);
200     if (AreEqual(storedLinkValue.Source, value) ||
201         ↪ AreEqual(storedLinkValue.Target, value))
202     {
203         return GetOne();
204     }
205     return GetZero();
206 }
207 if (restrictions.Count == 3)
208 {
209     var externalReferencesRange = constants.ExternalReferencesRange;
210     var source = restrictions[constants.SourcePart];
211     var target = restrictions[constants.TargetPart];
212     if (AreEqual(index, any))
213     {
214         if (AreEqual(source, any) && AreEqual(target, any))
215         {
216             return Total;
217         }
218         else if (AreEqual(source, any))
219         {
220             if (externalReferencesRange.HasValue &&
221                 ↪ externalReferencesRange.Value.Contains(target))
222             {
223                 return ExternalTargetsTreeMethods.CountUsages(target);
224             }
225             else
226             {
227                 return InternalTargetsTreeMethods.CountUsages(target);
228             }
229         }
230         else if (AreEqual(target, any))
231         {
232             if (externalReferencesRange.HasValue &&
233                 ↪ externalReferencesRange.Value.Contains(source))
234             {
235                 return ExternalSourcesTreeMethods.CountUsages(source);
236             }
237             else
238             {
239                 if (_useLinkedList)
240                 {
241                     return InternalSourcesListMethods.CountUsages(source);
242                 }
243                 else
244                 {
245                     return InternalSourcesTreeMethods.CountUsages(source);
246                 }
247             }
248         }
249     }
250 }
251 else //if(source != Any && target != Any)
252 {
253     // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
254     TLink link;
255     if (externalReferencesRange.HasValue)
256     {
257         if (externalReferencesRange.Value.Contains(source) &&
258             ↪ externalReferencesRange.Value.Contains(target))
259         {
260             link = ExternalSourcesTreeMethods.Search(source, target);
261         }
262     }
263 }

```

```

256         else if (externalReferencesRange.Value.Contains(source))
257         {
258             link = InternalTargetsTreeMethods.Search(source, target);
259         }
260         else if (externalReferencesRange.Value.Contains(target))
261         {
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     }

```

```

328     }
329     return GetZero();
330 }
331 }
332 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ 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         var @continue = constants.Continue;
352         var any = constants.Any;
353         var index = restrictions[constants.IndexPart];
354         if (restrictions.Count == 1)
355         {
356             if (AreEqual(index, any))
357             {
358                 return Each(handler, Array.Empty<TLink>());
359             }
360             if (!Exists(index))
361             {
362                 return @continue;
363             }
364             return handler(GetLinkStruct(index));
365         }
366         if (restrictions.Count == 2)
367         {
368             var value = restrictions[1];
369             if (AreEqual(index, any))
370             {
371                 if (AreEqual(value, any))
372                 {
373                     return Each(handler, Array.Empty<TLink>());
374                 }
375                 if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
376                 {
377                     return @break;
378                 }
379                 return Each(handler, new Link<TLink>(index, any, value));
380             }
381             else
382             {
383                 if (!Exists(index))
384                 {
385                     return @continue;
386                 }
387                 if (AreEqual(value, any))
388                 {
389                     return handler(GetLinkStruct(index));
390                 }
391                 ref var storedLinkValue = ref GetLinkDataPartReference(index);
392                 if (AreEqual(storedLinkValue.Source, value) ||
393                     AreEqual(storedLinkValue.Target, value))
394                 {
395                     return handler(GetLinkStruct(index));
396                 }
397                 return @continue;
398             }
399         }
400         if (restrictions.Count == 3)
401         {
402             var externalReferencesRange = constants.ExternalReferencesRange;
403             var source = restrictions[constants.SourcePart];

```

```

404 var target = restrictions[constants.TargetPart];
405 if (AreEqual(index, any))
406 {
407     if (AreEqual(source, any) && AreEqual(target, any))
408     {
409         return Each(handler, Array.Empty<TLink>());
410     }
411     else if (AreEqual(source, any))
412     {
413         if (externalReferencesRange.HasValue &&
414             ↪ externalReferencesRange.Value.Contains(target))
415         {
416             return ExternalTargetsTreeMethods.EachUsage(target, handler);
417         }
418         else
419         {
420             return InternalTargetsTreeMethods.EachUsage(target, handler);
421         }
422     }
423     else if (AreEqual(target, any))
424     {
425         if (externalReferencesRange.HasValue &&
426             ↪ externalReferencesRange.Value.Contains(source))
427         {
428             return ExternalSourcesTreeMethods.EachUsage(source, handler);
429         }
430         else
431         {
432             if (_useLinkedList)
433             {
434                 return InternalSourcesListMethods.EachUsage(source, handler);
435             }
436             else
437             {
438                 return InternalSourcesTreeMethods.EachUsage(source, handler);
439             }
440         }
441     }
442     else //if(source != Any && target != Any)
443     {
444         TLink link;
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         {
478             if (_useLinkedList ||
479                 ↪ GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
480                 ↪ InternalTargetsTreeMethods.CountUsages(target)))
481             {
482                 link = InternalTargetsTreeMethods.Search(source, target);
483             }
484             else
485             {
486                 link = InternalSourcesTreeMethods.Search(source, target);
487             }
488             return AreEqual(link, constants.Null) ? @continue :
489                 ↪ handler(GetLinkStruct(link));
490         }
491     else
492     {
493         if (!Exists(index))
494         {
495             return @continue;
496         }
497         if (AreEqual(source, any) && AreEqual(target, any))
498         {
499             return handler(GetLinkStruct(index));
500         }
501         ref var storedLinkValue = ref GetLinkDataPartReference(index);
502         if (!AreEqual(source, any) && !AreEqual(target, any))
503         {
504             if (AreEqual(storedLinkValue.Source, source) &&
505                 AreEqual(storedLinkValue.Target, target))
506             {
507                 return handler(GetLinkStruct(index));
508             }
509             return @continue;
510         }
511         var value = default(TLink);
512         if (AreEqual(source, any))
513         {
514             value = target;
515         }
516         if (AreEqual(target, any))
517         {
518             value = source;
519         }
520         if (AreEqual(storedLinkValue.Source, value) ||
521             AreEqual(storedLinkValue.Target, value))
522         {
523             return handler(GetLinkStruct(index));
524         }
525         return @continue;
526     }
527 }
528 throw new NotSupportedException("Другие размеры и способы ограничений не
529     ↪ поддерживаются.");
530
531 /// <remarks>
532 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
533     ↪ в другом месте (но не в менеджере памяти, а в логике Links)
534 /// </remarks>
535 [MethodImpl(MethodImplOptions.AggressiveInlining)]
536 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
537 {
538     var constants = Constants;
539     var @null = constants.Null;
540     var externalReferencesRange = constants.ExternalReferencesRange;
541     var linkIndex = restrictions[constants.IndexPart];
542     ref var link = ref GetLinkDataPartReference(linkIndex);
543     var source = link.Source;
544     var target = link.Target;
545     ref var header = ref GetHeaderReference();
546     ref var rootAsSource = ref header.RootAsSource;
547     ref var rootAsTarget = ref header.RootAsTarget;
548     // Будет корректно работать только в том случае, если пространство выделенной связи
549     ↪ предварительно заполнено нулями
550     if (!AreEqual(source, @null))
551     {

```

```

549         if (externalReferencesRange.HasValue &&
550             ↪ externalReferencesRange.Value.Contains(source))
551         {
552             ExternalSourcesTreeMethods.Detach(ref rootAsSource, linkIndex);
553         }
554         else
555         {
556             if (_useLinkedList)
557             {
558                 InternalSourcesListMethods.Detach(source, linkIndex);
559             }
560             else
561             {
562                 InternalSourcesTreeMethods.Detach(ref
563                     ↪ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
564             }
565         }
566     }
567     if (!AreEqual(target, @null))
568     {
569         if (externalReferencesRange.HasValue &&
570             ↪ externalReferencesRange.Value.Contains(target))
571         {
572             ExternalTargetsTreeMethods.Detach(ref rootAsTarget, linkIndex);
573         }
574         else
575         {
576             InternalTargetsTreeMethods.Detach(ref
577                 ↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
578         }
579     }
580     source = link.Source = substitution[constants.SourcePart];
581     target = link.Target = substitution[constants.TargetPart];
582     if (!AreEqual(source, @null))
583     {
584         if (externalReferencesRange.HasValue &&
585             ↪ externalReferencesRange.Value.Contains(source))
586         {
587             ExternalSourcesTreeMethods.Attach(ref rootAsSource, linkIndex);
588         }
589         else
590         {
591             if (_useLinkedList)
592             {
593                 InternalSourcesListMethods.AttachAsLast(source, linkIndex);
594             }
595             else
596             {
597                 InternalSourcesTreeMethods.Attach(ref
598                     ↪ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
599             }
600         }
601     }
602     if (!AreEqual(target, @null))
603     {
604         if (externalReferencesRange.HasValue &&
605             ↪ externalReferencesRange.Value.Contains(target))
606         {
607             ExternalTargetsTreeMethods.Attach(ref rootAsTarget, linkIndex);
608         }
609         else
610         {
611             InternalTargetsTreeMethods.Attach(ref
612                 ↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
613         }
614     }
615     return linkIndex;
616 }
617
618 /// <remarks>
619 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
620 ↪ пространство
621 /// </remarks>
622 [MethodImpl(MethodImplOptions.AggressiveInlining)]
623 public virtual TLink Create(ICollection<TLink> restrictions)
624 {
625     ref var header = ref GetHeaderReference();
626     var freeLink = header.FirstFreeLink;

```



```

618         if (!AreEqual(freeLink, Constants.Null))
619         {
620             UnusedLinksListMethods.Detach(freeLink);
621         }
622     else
623     {
624         var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
625         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 += _dataMemory.ReservationStepInBytes;
632             _indexMemory.ReservedCapacity += _indexMemory.ReservationStepInBytes;
633             SetPointers(_dataMemory, _indexMemory);
634             header = ref GetHeaderReference();
635             header.ReservedLinks = ConvertToAddress(_dataMemory.ReservedCapacity /
636                 ↳ LinkDataPartSizeInBytes);
637         }
638         freeLink = header.AllocatedLinks = Increment(header.AllocatedLinks);
639         _dataMemory.UsedCapacity += LinkDataPartSizeInBytes;
640         _indexMemory.UsedCapacity += LinkIndexPartSizeInBytes;
641     }
642     return freeLink;
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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
672 public IList<TLink> GetLinkStruct(TLink linkIndex)
673 {
674     ref var link = ref GetLinkDataPartReference(linkIndex);
675     return new Link<TLink>(linkIndex, link.Source, link.Target);
676 }
677 /// <remarks>
678 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
679 ↳ адрес реально поменялся
680 ///
681 /// Указатель this.links может быть в том же месте,
682 /// так как 0-я связь не используется и имеет такой же размер как Header,
683 /// поэтому header размещается в том же месте, что и 0-я связь
684 /// </remarks>
685 [MethodImpl(MethodImplOptions.AggressiveInlining)]
686 protected abstract void SetPointers(IResizableDirectMemory dataMemory,
687     ↳ IResizableDirectMemory indexMemory);
688 [MethodImpl(MethodImplOptions.AggressiveInlining)]
689 protected virtual void ResetPointers()
690 {
691     InternalSourcesListMethods = null;
692     InternalSourcesTreeMethods = null;

```

```

692     ExternalSourcesTreeMethods = null;
693     InternalTargetsTreeMethods = null;
694     ExternalTargetsTreeMethods = null;
695     UnusedLinksListMethods = null;
696 }
697
698 [MethodImpl(MethodImplOptions.AggressiveInlining)]
699 protected abstract ref LinksHeader<TLink> GetHeaderReference();
700
701 [MethodImpl(MethodImplOptions.AggressiveInlining)]
702 protected abstract ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex);
703
704 [MethodImpl(MethodImplOptions.AggressiveInlining)]
705 protected abstract ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
    ↪ 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
    ↪ is not needed
717     {
718         // TODO: Reduce access to memory in different location (should be enough to use
    ↪ just linkIndexPart)
719         ref var linkDataPart = ref GetLinkDataPartReference(linkIndex);
720         ref var linkIndexPart = ref GetLinkIndexPartReference(linkIndex);
721         return AreEqual(linkIndexPart.SizeAsTarget, default) &&
    ↪ !AreEqual(linkDataPart.Source, default);
722     }
723     else
724     {
725         return true;
726     }
727 }
728
729 [MethodImpl(MethodImplOptions.AggressiveInlining)]
730 protected virtual TLink GetOne() => _one;
731
732 [MethodImpl(MethodImplOptions.AggressiveInlining)]
733 protected virtual TLink GetZero() => default;
734
735 [MethodImpl(MethodImplOptions.AggressiveInlining)]
736 protected virtual bool AreEqual(TLink first, TLink second) =>
    ↪ _equalityComparer.Equals(first, second);
737
738 [MethodImpl(MethodImplOptions.AggressiveInlining)]
739 protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
    ↪ second) < 0;
740
741 [MethodImpl(MethodImplOptions.AggressiveInlining)]
742 protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
    ↪ _comparer.Compare(first, second) <= 0;
743
744 [MethodImpl(MethodImplOptions.AggressiveInlining)]
745 protected virtual bool GreaterThan(TLink first, TLink second) =>
    ↪ _comparer.Compare(first, second) > 0;
746
747 [MethodImpl(MethodImplOptions.AggressiveInlining)]
748 protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
    ↪ _comparer.Compare(first, second) >= 0;
749
750 [MethodImpl(MethodImplOptions.AggressiveInlining)]
751 protected virtual long ConvertToInt64(TLink value) =>
    ↪ _addressToInt64Converter.Convert(value);
752
753 [MethodImpl(MethodImplOptions.AggressiveInlining)]
754 protected virtual TLink ConvertToAddress(long value) =>
    ↪ _int64ToAddressConverter.Convert(value);
755
756 [MethodImpl(MethodImplOptions.AggressiveInlining)]
757 protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
    ↪ second);

```

```

758 [MethodImpl(MethodImplOptions.AggressiveInlining)]
759 protected virtual TLink Subtract(TLink first, TLink second) =>
760     ↪ Arithmetic<TLink>.Subtract(first, second);
761
762 [MethodImpl(MethodImplOptions.AggressiveInlining)]
763 protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
764
765 [MethodImpl(MethodImplOptions.AggressiveInlining)]
766 protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
767
768 #region Disposable
769
770 protected override bool AllowMultipleDisposeCalls
771 {
772     [MethodImpl(MethodImplOptions.AggressiveInlining)]
773     get => true;
774 }
775
776 [MethodImpl(MethodImplOptions.AggressiveInlining)]
777 protected override void Dispose(bool manual, bool wasDisposed)
778 {
779     if (!wasDisposed)
780     {
781         ResetPointers();
782         _dataMemory.DisposeIfPossible();
783         _indexMemory.DisposeIfPossible();
784     }
785 }
786
787 #endregion
788 }
789 }

```

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

```

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

```

```

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

```

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

```

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

```

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

```

1  using Platform.Unsafe;
2  using System;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Memory.Split
9  {
10     public struct RawLinkIndexPart<TLink> : IEquatable<RawLinkIndexPart<TLink>>

```

```

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

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Data.Doublets.Memory.Split.Generic;
3 using TLink = System.UInt32;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Memory.Split.Specific
8 {
9     public unsafe abstract class UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase :
10     ↳ ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
11     {
12         protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
13         protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
14         protected new readonly LinksHeader<TLink>* Header;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected
18         ↳ UInt32ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
19         ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
20         ↳ linksIndexParts, LinksHeader<TLink>* header)
21         : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
22         {
23             LinksDataParts = linksDataParts;
24             LinksIndexParts = linksIndexParts;
25             Header = header;
26         }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override TLink GetZero() => 0U;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override bool EqualToZero(TLink value) => value == 0U;
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected override bool AreEqual(TLink first, TLink second) => first == second;
36     }
37 }

```

```

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

```

1.50 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSizeBalancedTreeMethodsBase

```

1 using System.Runtime.CompilerServices;
2 using Platform.Data.Doublets.Memory.Split.Generic;
3 using TLink = System.UInt32;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Memory.Split.Specific

```

```

8 {
9     public unsafe abstract class UInt32ExternalLinksSizeBalancedTreeMethodsBase :
    ↳ ExternalLinksSizeBalancedTreeMethodsBase<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 UInt32ExternalLinksSizeBalancedTreeMethodsBase(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 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
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
88     {
89         ref var firstLink = ref LinksDataParts[first];
90         ref var secondLink = ref LinksDataParts[second];
91         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
            ↪ secondLink.Source, secondLink.Target);
92     }
93 }
94 }

```

1.51 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/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)]

```



```

48     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
49         ↪ TLink secondSource, TLink secondTarget)
50         => firstSource > secondSource || firstSource == secondSource && firstTarget >
51         ↪ secondTarget;
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override void ClearNode(TLink node)
55     {
56         ref var link = ref LinksIndexParts[node];
57         link.LeftAsSource = Zero;
58         link.RightAsSource = Zero;
59         link.SizeAsSource = Zero;
60     }
61 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt32;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      public unsafe class UInt32ExternalLinksSourcesSizeBalancedTreeMethods :
9          ↪ UInt32ExternalLinksSizeBalancedTreeMethodsBase
10      {
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          public UInt32ExternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
13              ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
14              ↪ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
15              ↪ linksIndexParts, header) { }
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override ref TLink GetLeftReference(TLink node) => ref
19              ↪ LinksIndexParts[node].LeftAsSource;
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          protected override ref TLink GetRightReference(TLink node) => ref
23              ↪ LinksIndexParts[node].RightAsSource;
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
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  }

```

```

50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override void ClearNode(TLink node)
53     {
54         ref var link = ref LinksIndexParts[node];
55         link.LeftAsSource = Zero;
56         link.RightAsSource = Zero;
57         link.SizeAsSource = Zero;
58     }
59 }
60 }

```

1.53 ./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 FirstIsToTheLeftOfSecond(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)

```

```

53     {
54         ref var link = ref LinksIndexParts[node];
55         link.LeftAsTarget = Zero;
56         link.RightAsTarget = Zero;
57         link.SizeAsTarget = Zero;
58     }
59 }
60 }

```

1.54 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksTargetsSizeBalancedTreeMethods

```

1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt32;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      public unsafe class UInt32ExternalLinksTargetsSizeBalancedTreeMethods :
9          ↳ UInt32ExternalLinksSizeBalancedTreeMethodsBase
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public UInt32ExternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink>
13             ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
14             ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
15             ↳ linksIndexParts, header) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected override ref TLink GetLeftReference(TLink node) => ref
19             ↳ LinksIndexParts[node].LeftAsTarget;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override ref TLink GetRightReference(TLink node) => ref
23             ↳ LinksIndexParts[node].RightAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsTarget;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsTarget;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetLeft(TLink node, TLink left) =>
33             ↳ LinksIndexParts[node].LeftAsTarget = left;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override void SetRight(TLink node, TLink right) =>
37             ↳ LinksIndexParts[node].RightAsTarget = right;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override void SetSize(TLink node, TLink size) =>
44             ↳ LinksIndexParts[node].SizeAsTarget = size;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override TLink GetTreeRoot() => Header->RootAsTarget;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
54             ↳ TLink secondSource, TLink secondTarget)
55             => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
56             ↳ secondSource;
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
60             ↳ TLink secondSource, TLink secondTarget)
61             => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
62             ↳ secondSource;
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected override void ClearNode(TLink node)
66         {
67             ref var link = ref LinksIndexParts[node];
68             link.LeftAsTarget = Zero;
69             link.RightAsTarget = Zero;
70         }
71     }
72 }

```

```

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

```

1.55 ./csharp/Platform.Data.Doublets.Memory.Split.Specific/UInt32InternalLinksRecursionlessSizeBalancedTree

```

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 bool LessThanZero(TLink value) => false; // value < 0 is always false
59     ↳ for ulong
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     protected override bool LessThan(TLink first, TLink second) => first < second;
63
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     protected override TLink Increment(TLink value) => ++value;
66
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     protected override TLink Decrement(TLink value) => --value;
69
70     [MethodImpl(MethodImplOptions.AggressiveInlining)]
71     protected override TLink Add(TLink first, TLink second) => first + second;
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     protected override TLink Subtract(TLink first, TLink second) => first - second;

```

```

68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
70         ↪ ref LinksDataParts[link];
71
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 }

```

1.56 ./csharp/Platform.Data.Doublets.Memory.Split.Specific/UInt32InternalLinksSizeBalancedTreeMethodsBase.

```

1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.Split.Generic;
3  using TLink = System.UInt32;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Memory.Split.Specific
8  {
9      public unsafe abstract class UInt32InternalLinksSizeBalancedTreeMethodsBase :
10         ↪ InternalLinksSizeBalancedTreeMethodsBase<TLink>
11     {
12         protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
13         protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
14         protected new readonly LinksHeader<TLink>* Header;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected UInt32InternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink>
18             ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
19             ↪ linksIndexParts, LinksHeader<TLink>* header)
20             : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
21         {
22             LinksDataParts = linksDataParts;
23             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)]

```

```

55     protected override bool LessThan(TLink first, TLink second) => first < second;
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override TLink Increment(TLink value) => ++value;
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected override TLink Decrement(TLink value) => --value;
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override TLink Add(TLink first, TLink second) => first + second;
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override TLink Subtract(TLink first, TLink second) => first - second;
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
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 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt32;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Generic
7  {
8      public unsafe class UInt32InternalLinksSourcesLinkedListMethods :
9          ↪ InternalLinksSourcesLinkedListMethods<TLink>
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 }

```

1.58 ./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)]

```

```

11     public
12         ↳ UInt32InternalLinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
13         ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
14         ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
15         ↳ linksIndexParts, header) { }
16
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     protected override ref TLink GetLeftReference(TLink node) => ref
19         ↳ LinksIndexParts[node].LeftAsSource;
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     protected override ref TLink GetRightReference(TLink node) => ref
23         ↳ LinksIndexParts[node].RightAsSource;
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override void SetLeft(TLink node, TLink left) =>
33         ↳ LinksIndexParts[node].LeftAsSource = left;
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override void SetRight(TLink node, TLink right) =>
37         ↳ LinksIndexParts[node].RightAsSource = right;
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override void SetSize(TLink node, TLink size) =>
44         ↳ LinksIndexParts[node].SizeAsSource = size;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override TLink GetTreeRoot(TLink node) => LinksIndexParts[node].RootAsSource;
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Target;
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override void ClearNode(TLink node)
57     {
58         ref var link = ref LinksIndexParts[node];
59         link.LeftAsSource = Zero;
60         link.RightAsSource = Zero;
61         link.SizeAsSource = Zero;
62     }
63
64     public override TLink Search(TLink source, TLink target) =>
65         ↳ SearchCore(GetTreeRoot(source), target);
66 }
67 }

```

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

```

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

```

```

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 }

```

1.60 ./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
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override void ClearNode(TLink node)
42         {
43             ref var link = ref LinksIndexParts[node];
44             link.LeftAsTarget = Zero;
45             link.RightAsTarget = Zero;
46         }
47
48         public override TLink Search(TLink source, TLink target) =>
49             SearchCore(GetTreeRoot(source), target);
50     }
51 }

```



```

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 }

```

1.61 ./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)]

```

```

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 }

```

1.62 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32SplitMemoryLinks.cs

```

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

```

```

37         _createExternalSourceTreeMethods = () => new
38         ↪ UInt32ExternalLinksSourcesSizeBalancedTreeMethods(Constants,
39         ↪ _linksDataParts, _linksIndexParts, _header);
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)]

```

```

96     protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
97
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     protected override bool GreaterThan(TLink first, TLink second) => first > second;
100
101     [MethodImpl(MethodImplOptions.AggressiveInlining)]
102     protected override bool GreaterOrEqualThan(TLink first, TLink second) => first >= second;
103
104     [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 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.Split.Generic;
3  using TLink = System.UInt32;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Memory.Split.Specific
8  {
9      public unsafe class UInt32UnusedLinksListMethods : UnusedLinksListMethods<TLink>
10     {
11         private readonly RawLinkDataPart<TLink>* _links;
12         private readonly LinksHeader<TLink>* _header;
13
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 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.Split.Generic;
3  using TLink = System.UInt64;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Memory.Split.Specific
8  {
9      public unsafe abstract class UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase :
10         ↪ ExternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
11     {
12         protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
13         protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
14     }
15 }

```

```

13     protected new readonly LinksHeader<TLink>* Header;
14
15     [MethodImpl(MethodImplOptions.AggressiveInlining)]
16     protected
17     ↪ UInt64ExternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
18     ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
19     ↪ linksIndexParts, LinksHeader<TLink>* header)
20     : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
21     {
22         LinksDataParts = linksDataParts;
23         LinksIndexParts = linksIndexParts;
24         Header = header;
25     }
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     protected override ulong GetZero() => 0UL;
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     protected override bool EqualToZero(ulong value) => value == 0UL;
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override bool AreEqual(ulong first, ulong second) => first == second;
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected override bool GreaterThanZero(ulong value) => value > 0UL;
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected override bool GreaterThan(ulong first, ulong second) => first > second;
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
47     ↪ always true for ulong
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
51     ↪ always >= 0 for ulong
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
58     ↪ for ulong
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected override bool LessThan(ulong first, ulong second) => first < second;
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override ulong Increment(ulong value) => ++value;
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override ulong Decrement(ulong value) => --value;
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override ulong Add(ulong first, ulong second) => first + second;
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     protected override ulong Subtract(ulong first, ulong second) => first - second;
74
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     protected override ref LinksHeader<TLink> GetHeaderReference() => ref *Header;
77
78     [MethodImpl(MethodImplOptions.AggressiveInlining)]
79     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
80     ↪ ref LinksDataParts[link];
81
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
84     ↪ ref LinksIndexParts[link];
85
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
88     {
89         ref var firstLink = ref LinksDataParts[first];
90         ref var secondLink = ref LinksDataParts[second];
91         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
92         ↪ secondLink.Source, secondLink.Target);
93     }

```

```

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 }

```

1.65 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSizeBalancedTreeMethodsBase

```

1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.Split.Generic;
3  using TLink = System.UInt64;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Memory.Split.Specific
8  {
9      public unsafe abstract class UInt64ExternalLinksSizeBalancedTreeMethodsBase :
10         ↪ ExternalLinksSizeBalancedTreeMethodsBase<TLink>, ILinksTreeMethods<TLink>
11     {
12         protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
13         protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
14         protected new readonly LinksHeader<TLink>* Header;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected UInt64ExternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink>
18             ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
19             ↪ linksIndexParts, LinksHeader<TLink>* header)
20             : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
21         {
22             LinksDataParts = linksDataParts;
23             LinksIndexParts = linksIndexParts;
24             Header = header;
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override ulong GetZero() => 0UL;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override bool EqualToZero(ulong value) => value == 0UL;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override bool AreEqual(ulong first, ulong second) => first == second;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override bool GreaterThanZero(ulong value) => value > 0UL;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override bool GreaterThan(ulong first, ulong second) => first > second;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
47             ↪ always true for ulong
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
51             ↪ always >= 0 for ulong
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
58             ↪ for ulong
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         protected override bool LessThan(ulong first, ulong second) => first < second;
62
63         [MethodImpl(MethodImplOptions.AggressiveInlining)]
64         protected override ulong Increment(ulong value) => ++value;
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

61     protected override ulong Decrement(ulong value) => --value;
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override ulong Add(ulong first, ulong second) => first + second;
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override ulong Subtract(ulong first, ulong second) => first - second;
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override ref LinksHeader<TLink> GetHeaderReference() => ref *Header;
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
74         ↪ ref LinksDataParts[link];
75
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
78         ↪ ref LinksIndexParts[link];
79
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
82     {
83         ref var firstLink = ref LinksDataParts[first];
84         ref var secondLink = ref LinksDataParts[second];
85         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
86             ↪ secondLink.Source, secondLink.Target);
87     }
88
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
91     {
92         ref var firstLink = ref LinksDataParts[first];
93         ref var secondLink = ref LinksDataParts[second];
94         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
95             ↪ secondLink.Source, secondLink.Target);
96     }
97 }
98
99 }

```

1.66 ./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 }

```

```

31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
33
34 [MethodImpl(MethodImplOptions.AggressiveInlining)]
35 protected override void SetSize(TLink node, TLink size) =>
36     ↳ LinksIndexParts[node].SizeAsSource = size;
37
38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 protected override TLink GetTreeRoot() => Header->RootAsSource;
40
41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
43
44 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 protected override bool FirstIsToLeftOfSecond(TLink firstSource, TLink firstTarget,
46     ↳ TLink secondSource, TLink secondTarget)
47     => firstSource < secondSource || firstSource == secondSource && firstTarget <
48     ↳ secondTarget;
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
52     ↳ TLink secondSource, TLink secondTarget)
53     => firstSource > secondSource || firstSource == secondSource && firstTarget >
54     ↳ secondTarget;
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 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using TLink = System.UInt64;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Memory.Split.Specific
7 {
8     public unsafe class UInt64ExternalLinksSourcesSizeBalancedTreeMethods :
9         ↳ UInt64ExternalLinksSizeBalancedTreeMethodsBase
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)]

```



```

35     protected override void SetSize(TLink node, TLink size) =>
36         ↳ LinksIndexParts[node].SizeAsSource = size;
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override TLink GetTreeRoot() => Header->RootAsSource;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
46         ↳ TLink secondSource, TLink secondTarget)
47         => firstSource < secondSource || firstSource == secondSource && firstTarget <
48         ↳ secondTarget;
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
52         ↳ TLink secondSource, TLink secondTarget)
53         => firstSource > secondSource || firstSource == secondSource && firstTarget >
54         ↳ secondTarget;
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 }

```

1.68 ./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 }

```

```

37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override TLink GetTreeRoot() => Header->RootAsTarget;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override bool 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 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt64;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      public unsafe class UInt64ExternalLinksTargetsSizeBalancedTreeMethods :
9          ↪ UInt64ExternalLinksSizeBalancedTreeMethodsBase
10      {
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          public UInt64ExternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink>
13              ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
14              ↪ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
15              ↪ linksIndexParts, header) { }
16
17          [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)]

```

```

41     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override bool FirstIsToTheLeftOfSecond(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 }

```

1.70 ./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 :
10         ↪ InternalLinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
11     {
12         protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
13         protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
14         protected new readonly LinksHeader<TLink>* Header;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected
18         ↪ UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<TLink>
19         ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
20         ↪ linksIndexParts, LinksHeader<TLink>* header)
21             : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts, (byte*)header)
22         {
23             LinksDataParts = linksDataParts;
24             LinksIndexParts = linksIndexParts;
25             Header = header;
26         }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override ulong GetZero() => OUL;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override bool EqualToZero(ulong value) => value == OUL;
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected override bool AreEqual(ulong first, ulong second) => first == second;
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected override bool GreaterThanZero(ulong value) => value > OUL;
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override bool GreaterThan(ulong first, ulong second) => first > second;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
48         ↪ always true for ulong
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override bool LessOrEqualThanZero(ulong value) => value == OUL; // value is
52         ↪ always >= 0 for ulong
53
54         [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) =>
    ↪ ref LinksDataParts[link];
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
    ↪ ref LinksIndexParts[link];
74
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     protected override bool FirstIsToLeftOfSecond(TLink first, TLink second) =>
    ↪ GetKeyPartValue(first) < GetKeyPartValue(second);
77
78     [MethodImpl(MethodImplOptions.AggressiveInlining)]
79     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
    ↪ GetKeyPartValue(first) > GetKeyPartValue(second);
80 }
81 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.Split.Generic;
3  using TLink = System.UInt64;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Memory.Split.Specific
8  {
9      public unsafe abstract class UInt64InternalLinksSizeBalancedTreeMethodsBase :
    ↪ InternalLinksSizeBalancedTreeMethodsBase<TLink>
10     {
11         protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
12         protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
13         protected new readonly LinksHeader<TLink>* Header;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected UInt64InternalLinksSizeBalancedTreeMethodsBase(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() => 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     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
43     ↪ always true for ulong
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
47     ↪ always >= 0 for ulong
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
54     ↪ for ulong
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected override bool LessThan(ulong first, ulong second) => first < second;
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override ulong Increment(ulong value) => ++value;
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected override ulong Decrement(ulong value) => --value;
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override ulong Add(ulong first, ulong second) => first + second;
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     protected override ulong Subtract(ulong first, ulong second) => first - second;
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
73     ↪ ref LinksDataParts[link];
74
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
77     ↪ ref LinksIndexParts[link];
78
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     protected override bool FirstIsToLeftOfSecond(TLink first, TLink second) =>
81     ↪ GetKeyPartValue(first) < GetKeyPartValue(second);
82
83     [MethodImpl(MethodImplOptions.AggressiveInlining)]
84     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
85     ↪ GetKeyPartValue(first) > GetKeyPartValue(second);
86 }
87 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt64;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Generic
7  {
8      public unsafe class UInt64InternalLinksSourcesLinkedListMethods :
9      ↪ InternalLinksSourcesLinkedListMethods<TLink>
10      {
11          private readonly RawLinkDataPart<TLink>* _linksDataParts;
12          private readonly RawLinkIndexPart<TLink>* _linksIndexParts;
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          public UInt64InternalLinksSourcesLinkedListMethods(LinksConstants<TLink> constants,
16          ↪ RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>* linksIndexParts)
17          : base(constants, (byte*)linksDataParts, (byte*)linksIndexParts)
18          {
19              _linksDataParts = linksDataParts;
20              _linksIndexParts = linksIndexParts;
21          }
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
25          ↪ ref _linksDataParts[link];
26
27

```

```

24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
        ↪ ref _linksIndexParts[link];
26 }
27 }

```

1.73 ./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 }

```

1.74 ./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
10    {
11        [MethodImpl(MethodImplOptions.AggressiveInlining)]
12        public UInt64InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
13            ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
14            ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
15            ↳ linksIndexParts, header) { }
16
17        [MethodImpl(MethodImplOptions.AggressiveInlining)]
18        protected override ref TLink GetLeftReference(TLink node) => ref
19            ↳ LinksIndexParts[node].LeftAsSource;
20
21        [MethodImpl(MethodImplOptions.AggressiveInlining)]
22        protected override ref TLink GetRightReference(TLink node) => ref
23            ↳ LinksIndexParts[node].RightAsSource;
24
25        [MethodImpl(MethodImplOptions.AggressiveInlining)]
26        protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
27
28        [MethodImpl(MethodImplOptions.AggressiveInlining)]
29        protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
30
31        [MethodImpl(MethodImplOptions.AggressiveInlining)]
32        protected override void SetLeft(TLink node, TLink left) =>
33            ↳ LinksIndexParts[node].LeftAsSource = left;
34
35        [MethodImpl(MethodImplOptions.AggressiveInlining)]
36        protected override void SetRight(TLink node, TLink right) =>
37            ↳ LinksIndexParts[node].RightAsSource = right;
38
39        [MethodImpl(MethodImplOptions.AggressiveInlining)]
40        protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
41
42        [MethodImpl(MethodImplOptions.AggressiveInlining)]
43        protected override void SetSize(TLink node, TLink size) =>
44            ↳ LinksIndexParts[node].SizeAsSource = size;
45
46        [MethodImpl(MethodImplOptions.AggressiveInlining)]
47        protected override TLink GetTreeRoot(TLink node) => LinksIndexParts[node].RootAsSource;
48
49        [MethodImpl(MethodImplOptions.AggressiveInlining)]
50        protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
51
52        [MethodImpl(MethodImplOptions.AggressiveInlining)]
53        protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Target;
54
55        [MethodImpl(MethodImplOptions.AggressiveInlining)]
56        protected override void ClearNode(TLink node)
57        {
58            ref var link = ref LinksIndexParts[node];
59            link.LeftAsSource = Zero;
60            link.RightAsSource = Zero;
61            link.SizeAsSource = Zero;
62        }
63
64        public override TLink Search(TLink source, TLink target) =>
65            ↳ SearchCore(GetTreeRoot(source), target);
66    }
67 }

```

1.75 ./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 :
9         ↳ UInt64InternalLinksRecursionlessSizeBalancedTreeMethodsBase
10    {
11        [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

11     public
12         ↳ UInt64InternalLinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink>
13         ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
14         ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
15         ↳ linksIndexParts, header) { }
16
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     protected override ref ulong GetLeftReference(ulong node) => ref
19         ↳ LinksIndexParts[node].LeftAsTarget;
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     protected override ref ulong GetRightReference(ulong node) => ref
23         ↳ LinksIndexParts[node].RightAsTarget;
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsTarget;
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsTarget;
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override void SetLeft(TLink node, TLink left) =>
33         ↳ LinksIndexParts[node].LeftAsTarget = left;
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override void SetRight(TLink node, TLink right) =>
37         ↳ LinksIndexParts[node].RightAsTarget = right;
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override void SetSize(TLink node, TLink size) =>
44         ↳ LinksIndexParts[node].SizeAsTarget = size;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override TLink GetTreeRoot(TLink node) => LinksIndexParts[node].RootAsTarget;
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Source;
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override void ClearNode(TLink node)
57     {
58         ref var link = ref LinksIndexParts[node];
59         link.LeftAsTarget = Zero;
60         link.RightAsTarget = Zero;
61         link.SizeAsTarget = Zero;
62     }
63
64     public override TLink Search(TLink source, TLink target) =>
65         ↳ SearchCore(GetTreeRoot(target), source);
66 }
67 }

```

1.76 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksTargetsSizeBalancedTreeMetho

```

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
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```



```

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

```

1.77 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64SplitMemoryLinks.cs

```

1 using System;
2 using System.Runtime.CompilerServices;
3 using Platform.Singletons;
4 using Platform.Memory;
5 using Platform.Data.Doublets.Memory.Split.Generic;
6 using TLink = System.UInt64;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Memory.Split.Specific
11 {
12     public unsafe class UInt64SplitMemoryLinks : SplitMemoryLinksBase<TLink>
13     {
14         private readonly Func<ILinksTreeMethods<TLink>> _createInternalSourceTreeMethods;
15         private readonly Func<ILinksTreeMethods<TLink>> _createExternalSourceTreeMethods;
16         private readonly Func<ILinksTreeMethods<TLink>> _createInternalTargetTreeMethods;
17         private readonly Func<ILinksTreeMethods<TLink>> _createExternalTargetTreeMethods;
18         private LinksHeader<ulong>* _header;
19         private RawLinkDataPart<ulong>* _linksDataParts;
20         private RawLinkIndexPart<ulong>* _linksIndexParts;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↳ indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
            ↳ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
            ↳ memoryReservationStep, Default<LinksConstants<TLink>>.Instance,
            ↳ IndexTreeType.Default, useLinkedList: true) { }
27

```

```

28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
    ↳ this(dataMemory, indexMemory, memoryReservationStep, constants,
    ↳ IndexTreeType.Default, useLinkedList: true) { }

30
31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 public 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)
85     => ref _linksDataParts[linkIndex];
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
89     linkIndex) => ref _linksIndexParts[linkIndex];
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 protected override bool AreEqual(ulong first, ulong second) => first == second;
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 protected override bool LessThan(ulong first, ulong second) => first < second;
96
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 protected override bool GreaterThan(ulong first, ulong second) => first > second;
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 protected override ulong GetZero() => 0UL;
108
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 protected override ulong GetOne() => 1UL;
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 protected override long ConvertToInt64(ulong value) => (long)value;
114
115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 protected override ulong ConvertToAddress(long value) => (ulong)value;
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 protected override ulong Add(ulong first, ulong second) => first + second;
120
121 [MethodImpl(MethodImplOptions.AggressiveInlining)]
122 protected override ulong Subtract(ulong first, ulong second) => first - second;
123
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 protected override ulong Increment(ulong link) => ++link;
126
127 [MethodImpl(MethodImplOptions.AggressiveInlining)]
128 protected override ulong Decrement(ulong link) => --link;
129 }
130 }

```

1.78 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64UnusedLinksListMethods.cs

```

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

```

```

26         protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
27     }
28 }

```

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

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7  using Platform.Numbers;
8  using static System.Runtime.CompilerServices.Unsafe;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.Memory.United.Generic
13 {
14     public unsafe abstract class LinksAvlBalancedTreeMethodsBase<TLink> :
15         ↳ SizedAndThreadedAVLBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
16     {
17         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
18             ↳ UncheckedConverter<TLink, long>.Default;
19         private static readonly UncheckedConverter<TLink, int> _addressToInt32Converter =
20             ↳ UncheckedConverter<TLink, int>.Default;
21         private static readonly UncheckedConverter<bool, TLink> _boolToAddressConverter =
22             ↳ UncheckedConverter<bool, TLink>.Default;
23         private static readonly UncheckedConverter<TLink, bool> _addressToBoolConverter =
24             ↳ UncheckedConverter<TLink, bool>.Default;
25         private static readonly UncheckedConverter<int, TLink> _int32ToAddressConverter =
26             ↳ UncheckedConverter<int, TLink>.Default;
27
28         protected readonly TLink Break;
29         protected readonly TLink Continue;
30         protected readonly byte* Links;
31         protected readonly byte* Header;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
35             ↳ byte* header)
36         {
37             Links = links;
38             Header = header;
39             Break = constants.Break;
40             Continue = constants.Continue;
41         }
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected abstract TLink GetTreeRoot();
45
46         [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         {

```

```

64     ref var firstLink = ref GetLinkReference(first);
65     ref var secondLink = ref GetLinkReference(second);
66     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
    ↪     secondLink.Source, secondLink.Target);
67 }
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
71 {
72     ref var firstLink = ref GetLinkReference(first);
73     ref var secondLink = ref GetLinkReference(second);
74     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↪     secondLink.Source, secondLink.Target);
75 }
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected virtual TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5,
    ↪     -5);
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 protected virtual void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
    ↪     Bit<TLink>.PartialWrite(storedValue, size, 5, -5);
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 protected virtual bool GetLeftIsChildValue(TLink value)
85 {
86     unchecked
87     {
88         return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 4, 1));
89         //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
90     }
91 }
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 protected virtual void SetLeftIsChildValue(ref TLink storedValue, bool value)
95 {
96     unchecked
97     {
98         var previousValue = storedValue;
99         var modified = Bit<TLink>.PartialWrite(previousValue,
    ↪         _boolToAddressConverter.Convert(value), 4, 1);
100         storedValue = modified;
101     }
102 }
103
104 [MethodImpl(MethodImplOptions.AggressiveInlining)]
105 protected virtual bool GetRightIsChildValue(TLink value)
106 {
107     unchecked
108     {
109         return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 3, 1));
110         //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
111     }
112 }
113
114 [MethodImpl(MethodImplOptions.AggressiveInlining)]
115 protected virtual void SetRightIsChildValue(ref TLink storedValue, bool value)
116 {
117     unchecked
118     {
119         var previousValue = storedValue;
120         var modified = Bit<TLink>.PartialWrite(previousValue,
    ↪         _boolToAddressConverter.Convert(value), 3, 1);
121         storedValue = modified;
122     }
123 }
124
125 [MethodImpl(MethodImplOptions.AggressiveInlining)]
126 protected bool IsChild(TLink parent, TLink possibleChild)
127 {
128     var parentSize = GetSize(parent);
129     var childSize = GetSizeOrZero(possibleChild);
130     return GreaterThanZero(childSize) && LessOrEqualThan(childSize, parentSize);
131 }
132
133 [MethodImpl(MethodImplOptions.AggressiveInlining)]
134 protected virtual sbyte GetBalanceValue(TLink storedValue)
135 {
136     unchecked

```

```

137     {
138         var value = _addressToInt32Converter.Convert(Bit<TLink>.PartialRead(storedValue,
139             ↪ 0, 3));
140         value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
141             ↪ end of sbyte
142         return (sbyte)value;
143     }
144 }
145 [MethodImpl(MethodImplOptions.AggressiveInlining)]
146 protected virtual void SetBalanceValue(ref TLink storedValue, sbyte value)
147 {
148     unchecked
149     {
150         var packagedValue = _int32ToAddressConverter.Convert((byte)value >> 5 & 4 |
151             ↪ value & 3);
152         var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
153         storedValue = modified;
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 /// <summary>
187 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
188 /// ↪ (концом).
189 /// </summary>
190 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
191 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
192 /// <returns>Индекс искомой связи.</returns>
193 [MethodImpl(MethodImplOptions.AggressiveInlining)]
194 public TLink Search(TLink source, TLink target)
195 {
196     var root = GetTreeRoot();
197     while (!EqualToZero(root))
198     {
199         ref var rootLink = ref GetLinkReference(root);
200         var rootSource = rootLink.Source;
201         var rootTarget = rootLink.Target;
202         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
203             ↪ node.Key < root.Key
204         {
205             root = GetLeftOrDefault(root);
206         }
207         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
208             ↪ node.Key > root.Key
209         {
210             root = GetRightOrDefault(root);
211         }
212     }
213 }

```

```

208         else // node.Key == root.Key
209         {
210             return root;
211         }
212     }
213     return Zero;
214 }
215
216 // TODO: Return indices range instead of references count
217 [MethodImpl(MethodImplOptions.AggressiveInlining)]
218 public TLink CountUsages(TLink link)
219 {
220     var root = GetTreeRoot();
221     var total = GetSize(root);
222     var totalRightIgnore = Zero;
223     while (!EqualToZero(root))
224     {
225         var @base = GetBasePartValue(root);
226         if (LessOrEqualThan(@base, link))
227         {
228             root = GetRightOrDefault(root);
229         }
230         else
231         {
232             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
233             root = GetLeftOrDefault(root);
234         }
235     }
236     root = GetTreeRoot();
237     var totalLeftIgnore = Zero;
238     while (!EqualToZero(root))
239     {
240         var @base = GetBasePartValue(root);
241         if (GreaterOrEqualThan(@base, link))
242         {
243             root = GetLeftOrDefault(root);
244         }
245         else
246         {
247             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
248             root = GetRightOrDefault(root);
249         }
250     }
251     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
252 }
253
254 [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

```

```

287         return Break;
288     }
289     current = GetNext(current);
290     if (EqualToZero(current) || !AreEqual(GetBasePartValue(current), link))
291     {
292         break;
293     }
294 }
295 }
296 return Continue;
297 }
298
299 [MethodImpl(MethodImplOptions.AggressiveInlining)]
300 protected override void PrintNodeValue(TLink node, StringBuilder sb)
301 {
302     ref var link = ref GetLinkReference(node);
303     sb.Append(' ');
304     sb.Append(link.Source);
305     sb.Append('-');
306     sb.Append('>');
307     sb.Append(link.Target);
308 }
309 }
310 }

```

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

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7  using static System.Runtime.CompilerServices.Unsafe;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Memory.United.Generic
12 {
13     public unsafe abstract class LinksRecursionlessSizeBalancedTreeMethodsBase<TLink> :
14         ↳ RecursionlessSizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
15     {
16         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
17             ↳ UncheckedConverter<TLink, long>.Default;
18
19         protected readonly TLink Break;
20         protected readonly TLink Continue;
21         protected readonly byte* Links;
22         protected readonly byte* Header;
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)]

```



```

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 }

```

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

```

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

```

```

40 [MethodImpl(MethodImplOptions.AggressiveInlining)]
41 protected abstract bool FirstIsToLeftOfSecond(TLink source, TLink target, TLink
    ↳ rootSource, TLink rootTarget);
42
43 [MethodImpl(MethodImplOptions.AggressiveInlining)]
44 protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
    ↳ AsRef<LinksHeader<TLink>>(Header);
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
    ↳ AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
    ↳ _addressToInt64Converter.Convert(link)));
48
49 [MethodImpl(MethodImplOptions.AggressiveInlining)]
50 protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
51 {
52     ref var link = ref GetLinkReference(linkIndex);
53     return new Link<TLink>(linkIndex, link.Source, link.Target);
54 }
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 protected override bool FirstIsToLeftOfSecond(TLink first, TLink second)
58 {
59     ref var firstLink = ref GetLinkReference(first);
60     ref var secondLink = ref GetLinkReference(second);
61     return FirstIsToLeftOfSecond(firstLink.Source, firstLink.Target,
    ↳ secondLink.Source, secondLink.Target);
62 }
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
66 {
67     ref var firstLink = ref GetLinkReference(first);
68     ref var secondLink = ref GetLinkReference(second);
69     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↳ secondLink.Source, secondLink.Target);
70 }
71
72 public TLink this[TLink index]
73 {
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     get
76     {
77         var root = GetTreeRoot();
78         if (GreaterOrEqualThan(index, GetSize(root)))
79         {
80             return Zero;
81         }
82         while (!EqualToZero(root))
83         {
84             var left = GetLeftOrDefault(root);
85             var leftSize = GetSizeOrZero(left);
86             if (LessThan(index, leftSize))
87             {
88                 root = left;
89                 continue;
90             }
91             if (AreEqual(index, leftSize))
92             {
93                 return root;
94             }
95             root = GetRightOrDefault(root);
96             index = Subtract(index, Increment(leftSize));
97         }
98         return Zero; // TODO: Impossible situation exception (only if tree structure
    ↳ broken)
99     }
100 }
101
102 /// <summary>
103 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
    ↳ (концом).
104 /// </summary>
105 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
106 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
107 /// <returns>Индекс искомой связи.</returns>
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 public TLink Search(TLink source, TLink target)

```

```

110 {
111     var root = GetTreeRoot();
112     while (!EqualToZero(root))
113     {
114         ref var rootLink = ref GetLinkReference(root);
115         var rootSource = rootLink.Source;
116         var rootTarget = rootLink.Target;
117         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
118             ↪ node.Key < root.Key
119         {
120             root = GetLeftOrDefault(root);
121         }
122         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
123             ↪ node.Key > root.Key
124         {
125             root = GetRightOrDefault(root);
126         }
127         else // node.Key == root.Key
128         {
129             return root;
130         }
131     }
132     return Zero;
133 }
134 // TODO: Return indices range instead of references count
135 [MethodImpl(MethodImplOptions.AggressiveInlining)]
136 public TLink CountUsages(TLink link)
137 {
138     var root = GetTreeRoot();
139     var total = GetSize(root);
140     var totalRightIgnore = Zero;
141     while (!EqualToZero(root))
142     {
143         var @base = GetBasePartValue(root);
144         if (LessOrEqualThan(@base, link))
145         {
146             root = GetRightOrDefault(root);
147         }
148         else
149         {
150             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
151             root = GetLeftOrDefault(root);
152         }
153     }
154     root = GetTreeRoot();
155     var totalLeftIgnore = Zero;
156     while (!EqualToZero(root))
157     {
158         var @base = GetBasePartValue(root);
159         if (GreaterOrEqualThan(@base, link))
160         {
161             root = GetLeftOrDefault(root);
162         }
163         else
164         {
165             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
166             root = GetRightOrDefault(root);
167         }
168     }
169     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
170 }
171 [MethodImpl(MethodImplOptions.AggressiveInlining)]
172 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
173     ↪ EachUsageCore(@base, GetTreeRoot(), handler);
174 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
175     ↪ low-level MSIL stack.
176 [MethodImpl(MethodImplOptions.AggressiveInlining)]
177 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
178 {
179     var @continue = Continue;
180     if (EqualToZero(link))
181     {
182         return @continue;
183     }
184     var linkBasePart = GetBasePartValue(link);
185     var @break = Break;

```

```

185         if (GreaterThan(linkBasePart, @base))
186         {
187             if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
188             {
189                 return @break;
190             }
191         }
192         else if (LessThan(linkBasePart, @base))
193         {
194             if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
195             {
196                 return @break;
197             }
198         }
199         else //if (linkBasePart == @base)
200         {
201             if (AreEqual(handler(GetLinkValues(link)), @break))
202             {
203                 return @break;
204             }
205             if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
206             {
207                 return @break;
208             }
209             if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
210             {
211                 return @break;
212             }
213         }
214         return @continue;
215     }
216
217     [MethodImpl(MethodImplOptions.AggressiveInlining)]
218     protected override void PrintNodeValue(TLink node, StringBuilder sb)
219     {
220         ref var link = ref GetLinkReference(node);
221         sb.Append(' ');
222         sb.Append(link.Source);
223         sb.Append('-');
224         sb.Append('>');
225         sb.Append(link.Target);
226     }
227 }
228 }

```

1.82 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesAvlBalancedTreeMethods.cs

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Memory.United.Generic
6 {
7     public unsafe class LinksSourcesAvlBalancedTreeMethods<TLink> :
8     ↪ LinksAvlBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksSourcesAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
12         ↪ byte* header) : base(constants, links, header) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref TLink GetLeftReference(TLink node) => ref
16         ↪ GetLinkReference(node).LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref TLink GetRightReference(TLink node) => ref
20         ↪ GetLinkReference(node).RightAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(TLink node, TLink left) =>
30         ↪ GetLinkReference(node).LeftAsSource = left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

28     protected override void SetRight(TLink node, TLink right) =>
29         ↳ GetLinkReference(node).RightAsSource = right;
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override TLink GetSize(TLink node) =>
33         ↳ GetSizeValue(GetLinkReference(node).SizeAsSource);
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
37         ↳ GetLinkReference(node).SizeAsSource, size);
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected override bool GetLeftIsChild(TLink node) =>
41         ↳ GetLeftIsChildValue(GetLinkReference(node).SizeAsSource);
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override void SetLeftIsChild(TLink node, bool value) =>
45         ↳ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override bool GetRightIsChild(TLink node) =>
49         ↳ GetRightIsChildValue(GetLinkReference(node).SizeAsSource);
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override void SetRightIsChild(TLink node, bool value) =>
53         ↳ SetRightIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override sbyte GetBalance(TLink node) =>
57         ↳ GetBalanceValue(GetLinkReference(node).SizeAsSource);
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
61         ↳ GetLinkReference(node).SizeAsSource, value);
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
71         ↳ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
72         ↳ (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
73
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
76         ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
77         ↳ (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
78
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     protected override void ClearNode(TLink node)
81     {
82         ref var link = ref GetLinkReference(node);
83         link.LeftAsSource = Zero;
84         link.RightAsSource = Zero;
85         link.SizeAsSource = Zero;
86     }
87 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Memory.United.Generic
6 {
7     public unsafe class LinksSourcesRecursionlessSizeBalancedTreeMethods<TLink> :
8         ↳ LinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink> constants,
12             ↳ byte* links, byte* header) : base(constants, links, header) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref TLink GetLeftReference(TLink node) => ref
16             ↳ GetLinkReference(node).LeftAsSource;
17     }
18 }

```

```

14 [MethodImpl(MethodImplOptions.AggressiveInlining)]
15 protected override ref TLink GetRightReference(TLink node) => ref
16     ↳ GetLinkReference(node).RightAsSource;
17
18 [MethodImpl(MethodImplOptions.AggressiveInlining)]
19 protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
20
21 [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 FirstIsToLeftOfSecond(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 }
64 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Memory.United.Generic
6 {
7     public unsafe class LinksSourcesSizeBalancedTreeMethods<TLink> :
8         ↳ LinksSizeBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
12             ↳ byte* header) : base(constants, links, header) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref TLink GetLeftReference(TLink node) => ref
16             ↳ GetLinkReference(node).LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref TLink GetRightReference(TLink node) => ref
20             ↳ GetLinkReference(node).RightAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

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 }

```

1.85 ./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsAvlBalancedTreeMethods.cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.United.Generic
6  {
7      public unsafe class LinksTargetsAvlBalancedTreeMethods<TLink> :
8          ↪ LinksAvlBalancedTreeMethodsBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksTargetsAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
12             ↪ byte* header) : base(constants, links, header) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref TLink GetLeftReference(TLink node) => ref
16             ↪ GetLinkReference(node).LeftAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref TLink GetRightReference(TLink node) => ref
20             ↪ GetLinkReference(node).RightAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(TLink node, TLink left) =>
30             ↪ GetLinkReference(node).LeftAsTarget = left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(TLink node, TLink right) =>
34             ↪ GetLinkReference(node).RightAsTarget = right;
35     }
36 }

```

```

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

```

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

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.United.Generic
6  {
7      public unsafe class LinksTargetsRecursionlessSizeBalancedTreeMethods<TLink> :
8      ↪ LinksRecursionlessSizeBalancedTreeMethodsBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<TLink> constants,
12         ↪ byte* links, byte* header) : base(constants, links, header) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref TLink GetLeftReference(TLink node) => ref
16         ↪ GetLinkReference(node).LeftAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref TLink GetRightReference(TLink node) => ref
20         ↪ GetLinkReference(node).RightAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override void SetLeftReference(TLink node, ref TLink value) =>
24         ↪ SetLeftReferenceValue(ref GetLinkReference(node).LeftAsTarget, value);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override void SetRightReference(TLink node, ref TLink value) =>
28         ↪ SetRightReferenceValue(ref GetLinkReference(node).RightAsTarget, value);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override void ClearNode(TLink node)
32         {
33             ref var link = ref GetLinkReference(node);
34             link.LeftAsTarget = Zero;
35             link.RightAsTarget = Zero;
36             link.SizeAsTarget = Zero;
37         }
38     }
39 }

```

```

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

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Memory.United.Generic
6 {
7     public unsafe class LinksTargetsSizeBalancedTreeMethods<TLink> :
    ↪ LinksSizeBalancedTreeMethodsBase<TLink>
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public LinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
    ↪ byte* header) : base(constants, links, header) { }
11
12        [MethodImpl(MethodImplOptions.AggressiveInlining)]
13        protected override ref TLink GetLeftReference(TLink node) => ref
    ↪ GetLinkReference(node).LeftAsTarget;
14
15        [MethodImpl(MethodImplOptions.AggressiveInlining)]
16        protected override ref TLink GetRightReference(TLink node) => ref
    ↪ GetLinkReference(node).RightAsTarget;
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
20
21        [MethodImpl(MethodImplOptions.AggressiveInlining)]
22        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;

```

```

23     [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 }

```

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

```

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

```

```

31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep) :
    ↳ this(memory, memoryReservationStep, Default<LinksConstants<TLink>>.Instance,
    ↳ IndexTreeType.Default) { }
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep,
    ↳ LinksConstants<TLink> constants, IndexTreeType indexTreeType) : base(memory,
    ↳ memoryReservationStep, constants)
37 {
38     if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
39     {
40         _createSourceTreeMethods = () => new
41             ↳ LinksSourcesAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
42         _createTargetTreeMethods = () => new
43             ↳ LinksTargetsAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
44     }
45     else
46     {
47         _createSourceTreeMethods = () => new
48             ↳ LinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
49         _createTargetTreeMethods = () => new
50             ↳ LinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
51     }
52     Init(memory, memoryReservationStep);
53 }
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 protected override void SetPointers(IResizableDirectMemory memory)
57 {
58     _links = (byte*)memory.Pointer;
59     _header = _links;
60     SourcesTreeMethods = _createSourceTreeMethods();
61     TargetsTreeMethods = _createTargetTreeMethods();
62     UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_links, _header);
63 }
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override void ResetPointers()
67 {
68     base.ResetPointers();
69     _links = null;
70     _header = null;
71 }
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 protected override ref LinksHeader<TLink> GetHeaderReference() => ref
75     ↳ AsRef<LinksHeader<TLink>>(_header);
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected override ref RawLink<TLink> GetLinkReference(TLink linkIndex) => ref
79     ↳ AsRef<RawLink<TLink>>(_links + (LinkSizeInBytes * ConvertToInt64(linkIndex)));
80 }
81 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Disposables;
5 using Platform.Singletons;
6 using Platform.Converters;
7 using Platform.Numbers;
8 using Platform.Memory;
9 using Platform.Data.Exceptions;
10
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Data.Doublets.Memory.United.Generic
14 {
15     public abstract class UnitedMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16     {
17         private static readonly EqualityComparer<TLink> _equalityComparer =
18             ↳ EqualityComparer<TLink>.Default;
19         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
20         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
21             ↳ UncheckedConverter<TLink, long>.Default;

```

```

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,
    ↳ нужно использовать не список а дерево, так как так можно быстрее проверить на
    ↳ наличие связи внутри
42 protected ILinksListMethods<TLink> UnusedLinksListMethods;
43
44 /// <summary>
45 /// Возвращает общее число связей находящихся в хранилище.
46 /// </summary>
47 protected virtual TLink Total
48 {
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     get
51     {
52         ref var header = ref GetHeaderReference();
53         return Subtract(header.AllocatedLinks, header.FreeLinks);
54     }
55 }
56
57 public virtual LinksConstants<TLink> Constants
58 {
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     get;
61 }
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
    ↳ memoryReservationStep, LinksConstants<TLink> constants)
65 {
66     _memory = memory;
67     _memoryReservationStep = memoryReservationStep;
68     Constants = constants;
69 }
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
    ↳ memoryReservationStep) : this(memory, memoryReservationStep,
    ↳ Default<LinksConstants<TLink>>.Instance) { }
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
76 {
77     if (memory.ReservedCapacity < memoryReservationStep)
78     {
79         memory.ReservedCapacity = memoryReservationStep;
80     }
81     SetPointers(memory);
82     ref var header = ref GetHeaderReference();
83     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
84     memory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) * LinkSizeInBytes) +
        ↳ LinkHeaderSizeInBytes;
85     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
86     header.ReservedLinks = ConvertToAddress((memory.ReservedCapacity -
        ↳ LinkHeaderSizeInBytes) / LinkSizeInBytes);
87 }
88
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 public virtual TLink Count(IList<TLink> restrictions)
91 {

```

```

92 // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
93 if (restrictions.Count == 0)
94 {
95     return Total;
96 }
97 var constants = Constants;
98 any = constants.Any;
99 var index = restrictions[constants.IndexPart];
100 if (restrictions.Count == 1)
101 {
102     if (AreEqual(index, any))
103     {
104         return Total;
105     }
106     return Exists(index) ? GetOne() : GetZero();
107 }
108 if (restrictions.Count == 2)
109 {
110     var value = restrictions[1];
111     if (AreEqual(index, any))
112     {
113         if (AreEqual(value, any))
114         {
115             return Total; // Any - как отсутствие ограничения
116         }
117         return Add(SourcesTreeMethods.CountUsages(value),
118             ↪ TargetsTreeMethods.CountUsages(value));
119     }
120     else
121     {
122         if (!Exists(index))
123         {
124             return GetZero();
125         }
126         if (AreEqual(value, any))
127         {
128             return GetOne();
129         }
130         ref var storedLinkValue = ref GetLinkReference(index);
131         if (AreEqual(storedLinkValue.Source, value) ||
132             ↪ AreEqual(storedLinkValue.Target, value))
133         {
134             return GetOne();
135         }
136         return GetZero();
137     }
138 }
139 if (restrictions.Count == 3)
140 {
141     var source = restrictions[constants.SourcePart];
142     var target = restrictions[constants.TargetPart];
143     if (AreEqual(index, any))
144     {
145         if (AreEqual(source, any) && AreEqual(target, any))
146         {
147             return Total;
148         }
149         else if (AreEqual(source, any))
150         {
151             return TargetsTreeMethods.CountUsages(target);
152         }
153         else if (AreEqual(target, any))
154         {
155             return SourcesTreeMethods.CountUsages(source);
156         }
157         else //if(source != Any && target != Any)
158         {
159             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
160             var link = SourcesTreeMethods.Search(source, target);
161             return AreEqual(link, constants.Null) ? GetZero() : GetOne();
162         }
163     }
164     else
165     {
166         if (!Exists(index))
167         {
168             return GetZero();
169         }
170     }
171 }

```

```

168         if (AreEqual(source, any) && AreEqual(target, any))
169         {
170             return GetOne();
171         }
172         ref var storedLinkValue = ref GetLinkReference(index);
173         if (!AreEqual(source, any) && !AreEqual(target, any))
174         {
175             if (AreEqual(storedLinkValue.Source, source) &&
176                 ↪ AreEqual(storedLinkValue.Target, target))
177             {
178                 return GetOne();
179             }
180             return GetZero();
181         }
182         var value = default(TLink);
183         if (AreEqual(source, any))
184         {
185             value = target;
186         }
187         if (AreEqual(target, any))
188         {
189             value = source;
190         }
191         if (AreEqual(storedLinkValue.Source, value) ||
192             ↪ AreEqual(storedLinkValue.Target, value))
193         {
194             return GetOne();
195         }
196         return GetZero();
197     }
198 }
199
200 [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         return @break;
243     }
244     return Each(handler, new Link<TLink>(index, any, value));
245 }
246 else
247 {
248     if (!Exists(index))
249     {
250         return @continue;
251     }
252     if (AreEqual(value, any))
253     {
254         return handler(GetLinkStruct(index));
255     }
256     ref var storedLinkValue = ref GetLinkReference(index);
257     if (AreEqual(storedLinkValue.Source, value) ||
258         AreEqual(storedLinkValue.Target, value))
259     {
260         return handler(GetLinkStruct(index));
261     }
262     return @continue;
263 }
264 }
265 if (restrictions.Count == 3)
266 {
267     var source = restrictions[constants.SourcePart];
268     var target = restrictions[constants.TargetPart];
269     if (AreEqual(index, any))
270     {
271         if (AreEqual(source, any) && AreEqual(target, any))
272         {
273             return Each(handler, Array.Empty<TLink>());
274         }
275         else if (AreEqual(source, any))
276         {
277             return TargetsTreeMethods.EachUsage(target, handler);
278         }
279         else if (AreEqual(target, any))
280         {
281             return SourcesTreeMethods.EachUsage(source, handler);
282         }
283         else //if(source != Any && target != Any)
284         {
285             var link = SourcesTreeMethods.Search(source, target);
286             return AreEqual(link, constants.Null) ? @continue :
287                 ↪ handler(GetLinkStruct(link));
288         }
289     }
290     else
291     {
292         if (!Exists(index))
293         {
294             return @continue;
295         }
296         if (AreEqual(source, any) && AreEqual(target, any))
297         {
298             return handler(GetLinkStruct(index));
299         }
300         ref var storedLinkValue = ref GetLinkReference(index);
301         if (!AreEqual(source, any) && !AreEqual(target, any))
302         {
303             if (AreEqual(storedLinkValue.Source, source) &&
304                 AreEqual(storedLinkValue.Target, target))
305             {
306                 return handler(GetLinkStruct(index));
307             }
308             return @continue;
309         }
310         var value = default(TLink);
311         if (AreEqual(source, any))
312         {
313             value = target;
314         }
315         if (AreEqual(target, any))
316         {
317             value = source;
318         }
319         if (AreEqual(storedLinkValue.Source, value) ||

```

```

319         AreEqual(storedLinkValue.Target, value))
320     {
321         return handler(GetLinkStruct(index));
322     }
323     return @continue;
324 }
325 }
326 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
327 }
328
329 /// <remarks>
330 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↳ в другом месте (но не в менеджере памяти, а в логике Links)
331 /// </remarks>
332 [MethodImpl(MethodImplOptions.AggressiveInlining)]
333 public virtual TLink Update(ICollection<TLink> restrictions, ICollection<TLink> substitution)
334 {
335     var constants = Constants;
336     var @null = constants.Null;
337     var linkIndex = restrictions[constants.IndexPart];
338     ref var link = ref GetLinkReference(linkIndex);
339     ref var header = ref GetHeaderReference();
340     ref var firstAsSource = ref header.RootAsSource;
341     ref var firstAsTarget = ref header.RootAsTarget;
342     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↳ предварительно заполнено нулями
343     if (!AreEqual(link.Source, @null))
344     {
345         SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
346     }
347     if (!AreEqual(link.Target, @null))
348     {
349         TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
350     }
351     link.Source = substitution[constants.SourcePart];
352     link.Target = substitution[constants.TargetPart];
353     if (!AreEqual(link.Source, @null))
354     {
355         SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
356     }
357     if (!AreEqual(link.Target, @null))
358     {
359         TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
360     }
361     return linkIndex;
362 }
363
364 /// <remarks>
365 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
    ↳ пространство
366 /// </remarks>
367 [MethodImpl(MethodImplOptions.AggressiveInlining)]
368 public virtual TLink Create(ICollection<TLink> restrictions)
369 {
370     ref var header = ref GetHeaderReference();
371     var freeLink = header.FirstFreeLink;
372     if (!AreEqual(freeLink, Constants.Null))
373     {
374         UnusedLinksListMethods.Detach(freeLink);
375     }
376     else
377     {
378         var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
379         if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
380         {
381             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
382         }
383         if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
384         {
385             _memory.ReservedCapacity += _memory.ReservationStep;
386             SetPointers(_memory);
387             header = ref GetHeaderReference();
388             header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
    ↳ LinkSizeInBytes);
389         }
390         freeLink = header.AllocatedLinks = Increment(header.AllocatedLinks);
391         _memory.UsedCapacity += LinkSizeInBytes;
392     }

```

```

393     return freeLink;
394 }
395
396 [MethodImpl(MethodImplOptions.AggressiveInlining)]
397 public virtual void Delete(ICollection<TLink> restrictions)
398 {
399     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 ICollection<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 LinkHeader<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 }

```

```

468     }
469 }
470
471 [MethodImpl(MethodImplOptions.AggressiveInlining)]
472 protected virtual TLink GetOne() => _one;
473
474 [MethodImpl(MethodImplOptions.AggressiveInlining)]
475 protected virtual TLink GetZero() => default;
476
477 [MethodImpl(MethodImplOptions.AggressiveInlining)]
478 protected virtual bool AreEqual(TLink first, TLink second) =>
479     ↳ _equalityComparer.Equals(first, second);
480
481 [MethodImpl(MethodImplOptions.AggressiveInlining)]
482 protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
483     ↳ second) < 0;
484
485 [MethodImpl(MethodImplOptions.AggressiveInlining)]
486 protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
487     ↳ _comparer.Compare(first, second) <= 0;
488
489 [MethodImpl(MethodImplOptions.AggressiveInlining)]
490 protected virtual bool GreaterThan(TLink first, TLink second) =>
491     ↳ _comparer.Compare(first, second) > 0;
492
493 [MethodImpl(MethodImplOptions.AggressiveInlining)]
494 protected virtual 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 }

```

1.90 ./csharp/Platform.Data.Doublets/Memory/United/Generic/UnusedLinksListMethods.cs

```

1 using System.Runtime.CompilerServices;
2 using Platform.Collections.Methods.Lists;
3 using Platform.Converters;
4 using static System.Runtime.CompilerServices.Unsafe;
5

```

```

6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Memory.United.Generic
9 {
10     public unsafe class UnusedLinksListMethods<TLink> :
11         ↳ AbsoluteCircularDoublyLinkedListMethods<TLink>, ILinksListMethods<TLink>
12     {
13         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
14             ↳ UncheckedConverter<TLink, long>.Default;
15
16         private readonly byte* _links;
17         private readonly byte* _header;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnusedLinksListMethods(byte* links, byte* header)
21         {
22             _links = links;
23             _header = header;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
28             ↳ AsRef<LinksHeader<TLink>>(_header);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
32             ↳ AsRef<RawLink<TLink>>(_links + (RawLink<TLink>.SizeInBytes *
33             ↳ _addressToInt64Converter.Convert(link)));
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected override TLink GetPrevious(TLink element) => GetLinkReference(element).Source;
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override TLink GetNext(TLink element) => GetLinkReference(element).Target;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override TLink GetSize() => GetHeaderReference().FreeLinks;
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
52             ↳ element;
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
56             ↳ element;
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         protected override void SetPrevious(TLink element, TLink previous) =>
60             ↳ GetLinkReference(element).Source = previous;
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         protected override void SetNext(TLink element, TLink next) =>
64             ↳ GetLinkReference(element).Target = next;
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
68     }
69 }

```

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

```

1 using Platform.Unsafe;
2 using System;
3 using System.Collections.Generic;
4 using System.Runtime.CompilerServices;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Memory.United
9 {
10     public struct RawLink<TLink> : IEquatable<RawLink<TLink>>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↳ EqualityComparer<TLink>.Default;
14
15     }
16 }

```

```

14     public static readonly long SizeInBytes = Structure<RawLink<TLink>>.Size;
15
16     public TLink Source;
17     public TLink Target;
18     public TLink LeftAsSource;
19     public TLink RightAsSource;
20     public TLink SizeAsSource;
21     public TLink LeftAsTarget;
22     public TLink RightAsTarget;
23     public TLink SizeAsTarget;
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     public override bool Equals(object obj) => obj is RawLink<TLink> link ? Equals(link) :
        ↳ false;
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     public bool Equals(RawLink<TLink> other)
30         => _equalityComparer.Equals(Source, other.Source)
31         && _equalityComparer.Equals(Target, other.Target)
32         && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
33         && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
34         && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
35         && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
36         && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
37         && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     public override int GetHashCode() => (Source, Target, LeftAsSource, RightAsSource,
        ↳ SizeAsSource, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     public static bool operator ==(RawLink<TLink> left, RawLink<TLink> right) =>
        ↳ left.Equals(right);
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     public static bool operator !=(RawLink<TLink> left, RawLink<TLink> right) => !(left ==
        ↳ right);
47 }
48 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.United.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.United.Specific
7  {
8      public unsafe abstract class UInt32LinksRecursionlessSizeBalancedTreeMethodsBase :
9          ↳ LinksRecursionlessSizeBalancedTreeMethodsBase<uint>
10      {
11          protected new readonly RawLink<uint>* Links;
12          protected new readonly LinksHeader<uint>* Header;
13
14          protected UInt32LinksRecursionlessSizeBalancedTreeMethodsBase(LinksConstants<uint>
15              ↳ constants, RawLink<uint>* links, LinksHeader<uint>* header)
16              : base(constants, (byte*)links, (byte*)header)
17          {
18              Links = links;
19              Header = header;
20          }
21
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          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          protected override bool GreaterThanZero(uint value) => value > 0U;
33
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          protected override bool GreaterThan(uint first, uint second) => first > second;
36
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          protected override bool GreaterOrEqualThan(uint first, uint second) => first >= second;
39

```

```

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 }

```

1.93 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSizeBalancedTreeMethodsBase.cs

```

1 using System.Runtime.CompilerServices;
2 using Platform.Data.Doublets.Memory.United.Generic;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Memory.United.Specific
7 {
8     public unsafe abstract class UInt32LinksSizeBalancedTreeMethodsBase :
9     ↳ LinksSizeBalancedTreeMethodsBase<uint>
10     {
11         protected new readonly RawLink<uint>* Links;
12         protected new readonly LinksHeader<uint>* Header;
13
14         protected UInt32LinksSizeBalancedTreeMethodsBase(LinksConstants<uint> constants,
15     ↳ RawLink<uint>* links, LinksHeader<uint>* header)
16         : base(constants, (byte*)links, (byte*)header)
17         {
18             Links = links;
19             Header = header;
20         }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Memory.United.Specific

```



```

6 {
7     public unsafe class UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods :
8         ↳ UInt32LinksRecursionlessSizeBalancedTreeMethodsBase
9     {
10         public UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods(LinksConstants<uint>
11             ↳ constants, RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links,
12             ↳ header) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsSource;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected override ref uint GetRightReference(uint node) => ref
19             ↳ Links[node].RightAsSource;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected override uint GetLeft(uint node) => Links[node].LeftAsSource;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override uint GetRight(uint node) => Links[node].RightAsSource;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override void SetLeft(uint node, uint left) => Links[node].LeftAsSource = left;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected override void SetRight(uint node, uint right) => Links[node].RightAsSource =
32             ↳ right;
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected override uint GetSize(uint node) => Links[node].SizeAsSource;
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected override void SetSize(uint node, uint size) => Links[node].SizeAsSource = size;
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected override uint GetTreeRoot() => Header->RootAsSource;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override uint GetBasePartValue(uint link) => Links[link].Source;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override bool FirstIsToTheLeftOfSecond(uint firstSource, uint firstTarget,
48             ↳ uint secondSource, uint secondTarget)
49             => firstSource < secondSource || (firstSource == secondSource && firstTarget <
50             ↳ secondTarget);
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
54             ↳ uint secondSource, uint secondTarget)
55             => firstSource > secondSource || (firstSource == secondSource && firstTarget >
56             ↳ secondTarget);
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         protected override void ClearNode(uint node)
60         {
61             ref var link = ref Links[node];
62             link.LeftAsSource = 0U;
63             link.RightAsSource = 0U;
64             link.SizeAsSource = 0U;
65         }
66     }
67 }

```

1.95 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSourcesSizeBalancedTreeMethods.cs

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Memory.United.Specific
6 {
7     public unsafe class UInt32LinksSourcesSizeBalancedTreeMethods :
8         ↳ UInt32LinksSizeBalancedTreeMethodsBase
9     {
10         public UInt32LinksSourcesSizeBalancedTreeMethods(LinksConstants<uint> constants,
11             ↳ RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links, header) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsSource;
15
16

```

```

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

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Memory.United.Specific
6 {
7     public unsafe class UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods :
    ↳ UInt32LinksRecursionlessSizeBalancedTreeMethodsBase
8     {
9         public UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<uint>
    ↳ constants, RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links,
    ↳ header) { }
10
11 [MethodImpl(MethodImplOptions.AggressiveInlining)]
12 protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsTarget;
13
14 [MethodImpl(MethodImplOptions.AggressiveInlining)]
15 protected override ref uint GetRightReference(uint node) => ref
    ↳ Links[node].RightAsTarget;
16
17 [MethodImpl(MethodImplOptions.AggressiveInlining)]
18 protected override uint GetLeft(uint node) => Links[node].LeftAsTarget;
19
20 [MethodImpl(MethodImplOptions.AggressiveInlining)]
21 protected override uint GetRight(uint node) => Links[node].RightAsTarget;
22

```

```

23 [MethodImpl(MethodImplOptions.AggressiveInlining)]
24 protected override void SetLeft(uint node, uint left) => Links[node].LeftAsTarget = left;
25
26 [MethodImpl(MethodImplOptions.AggressiveInlining)]
27 protected override void SetRight(uint node, uint right) => Links[node].RightAsTarget =
    ↳ right;
28
29 [MethodImpl(MethodImplOptions.AggressiveInlining)]
30 protected override uint GetSize(uint node) => Links[node].SizeAsTarget;
31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 protected override void SetSize(uint node, uint size) => Links[node].SizeAsTarget = size;
34
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 FirstIsToLeftOfSecond(uint firstSource, uint firstTarget,
    ↳ uint secondSource, uint secondTarget)
    => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
    ↳ secondSource);
43
44 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
    ↳ uint secondSource, uint secondTarget)
    => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
    ↳ secondSource);
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 protected override void ClearNode(uint node)
49 {
50     {
51         ref var link = ref Links[node];
52         link.LeftAsTarget = 0U;
53         link.RightAsTarget = 0U;
54         link.SizeAsTarget = 0U;
55     }
56 }
57 }
58 }

```

1.97 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksTargetsSizeBalancedTreeMethods.cs

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Memory.United.Specific
6 {
7     public unsafe class UInt32LinksTargetsSizeBalancedTreeMethods :
8         ↳ UInt32LinksSizeBalancedTreeMethodsBase
9     {
10         public UInt32LinksTargetsSizeBalancedTreeMethods(LinksConstants<uint> constants,
11             ↳ RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links, header) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsTarget;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected override ref uint GetRightReference(uint node) => ref
18             ↳ Links[node].RightAsTarget;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override uint GetLeft(uint node) => Links[node].LeftAsTarget;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override uint GetRight(uint node) => Links[node].RightAsTarget;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override void SetLeft(uint node, uint left) => Links[node].LeftAsTarget = left;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override void SetRight(uint node, uint right) => Links[node].RightAsTarget =
31             ↳ right;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override uint GetSize(uint node) => Links[node].SizeAsTarget;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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 }

```

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

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Memory;
4  using Platform.Singletons;
5  using Platform.Data.Doublets.Memory.United.Generic;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Memory.United.Specific
10 {
11     /// <summary>
12     /// <para>Represents a low-level implementation of direct access to resizable memory, for
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) { }

```

```

38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
40     ↪ memoryReservationStep, LinksConstants<uint> constants, IndexTreeType indexTreeType)
41     ↪ : base(memory, memoryReservationStep, constants)
42 {
43     if (indexTreeType == IndexTreeType.SizeBalancedTree)
44     {
45         _createSourceTreeMethods = () => new
46             ↪ UInt32LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
47         _createTargetTreeMethods = () => new
48             ↪ UInt32LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
49     }
50     else
51     {
52         _createSourceTreeMethods = () => new
53             ↪ UInt32LinksSourcesRecursionlessSizeBalancedTreeMethods(Constants, _links,
54             ↪ _header);
55         _createTargetTreeMethods = () => new
56             ↪ UInt32LinksTargetsRecursionlessSizeBalancedTreeMethods(Constants, _links,
57             ↪ _header);
58     }
59     Init(memory, memoryReservationStep);
60 }
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override void SetPointers(IResizableDirectMemory memory)
64 {
65     _header = (LinksHeader<uint>*)memory.Pointer;
66     _links = (RawLink<uint>*)memory.Pointer;
67     SourcesTreeMethods = _createSourceTreeMethods();
68     TargetsTreeMethods = _createTargetTreeMethods();
69     UnusedLinksListMethods = new UInt32UnusedLinksListMethods(_links, _header);
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 protected override void ResetPointers()
74 {
75     base.ResetPointers();
76     _links = null;
77     _header = null;
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 protected override ref LinksHeader<uint> GetHeaderReference() => ref *_header;
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 protected override ref RawLink<uint> GetLinkReference(uint linkIndex) => ref
85     ↪ _links[linkIndex];
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 protected override bool AreEqual(uint first, uint second) => first == second;
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 protected override bool LessThan(uint first, uint second) => first < second;
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 protected override bool LessOrEqualThan(uint first, uint second) => first <= second;
95
96 [MethodImpl(MethodImplOptions.AggressiveInlining)]
97 protected override bool GreaterThan(uint first, uint second) => first > second;
98
99 [MethodImpl(MethodImplOptions.AggressiveInlining)]
100 protected override bool GreaterOrEqualThan(uint first, uint second) => first >= second;
101
102 [MethodImpl(MethodImplOptions.AggressiveInlining)]
103 protected override uint GetZero() => 0U;
104
105 [MethodImpl(MethodImplOptions.AggressiveInlining)]
106 protected override uint GetOne() => 1U;
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 protected override long ConvertToInt64(uint value) => (long)value;
110
111 [MethodImpl(MethodImplOptions.AggressiveInlining)]
112 protected override uint ConvertToAddress(long value) => (uint)value;
113
114 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

107     protected override uint Add(uint first, uint second) => first + second;
108
109     [MethodImpl(MethodImplOptions.AggressiveInlining)]
110     protected override uint Subtract(uint first, uint second) => first - second;
111
112     [MethodImpl(MethodImplOptions.AggressiveInlining)]
113     protected override uint Increment(uint link) => ++link;
114
115     [MethodImpl(MethodImplOptions.AggressiveInlining)]
116     protected override uint Decrement(uint link) => --link;
117 }
118 }

```

1.99 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32UnusedLinksListMethods.cs

```

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

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Data.Doublets.Memory.United.Generic;
3 using static System.Runtime.CompilerServices.Unsafe;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Memory.United.Specific
8 {
9     public unsafe abstract class UInt64LinksAvlBalancedTreeMethodsBase :
10         ↳ LinksAvlBalancedTreeMethodsBase<ulong>
11     {
12         protected new readonly RawLink<ulong>* Links;
13         protected new readonly LinksHeader<ulong>* Header;
14
15         protected UInt64LinksAvlBalancedTreeMethodsBase(LinksConstants<ulong> constants,
16             ↳ RawLink<ulong>* links, LinksHeader<ulong>* header)
17             : base(constants, (byte*)links, (byte*)header)
18         {
19             Links = links;
20             Header = header;
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override ulong GetZero() => OUL;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override bool EqualToZero(ulong value) => value == OUL;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override bool AreEqual(ulong first, ulong second) => first == second;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override bool GreaterThanZero(ulong value) => value > OUL;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override bool GreaterThan(ulong first, ulong second) => first > second;
37     }
38 }

```

```

36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
    ↳ always true for ulong
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
    ↳ always >= 0 for ulong
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
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) =>
105         ↪ storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
106         ↪ value & 3) & 7UL);
107
108     [MethodImpl(MethodImplOptions.AggressiveInlining)]
109     protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
110
111     [MethodImpl(MethodImplOptions.AggressiveInlining)]
112     protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
113 }
114 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.United.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.United.Specific
7  {
8      public unsafe abstract class UInt64LinksRecursionlessSizeBalancedTreeMethodsBase :
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
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetZero() => 0UL;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override bool EqualToZero(ulong value) => value == 0UL;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override bool AreEqual(ulong first, ulong second) => first == second;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override bool GreaterThanZero(ulong value) => value > 0UL;
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 == 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)]

```



```

63     protected override ulong Subtract(ulong first, ulong second) => first - second;
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
67     {
68         ref var firstLink = ref Links[first];
69         ref var secondLink = ref Links[second];
70         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
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 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.Memory.United.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.United.Specific
7  {
8      public unsafe abstract class UInt64LinksSizeBalancedTreeMethodsBase :
9          ↪ LinksSizeBalancedTreeMethodsBase<ulong>
10     {
11         protected new readonly RawLink<ulong>* Links;
12         protected new readonly LinksHeader<ulong>* Header;
13
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)]

```

```

48     protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
    ↪ for ulong
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override bool LessThan(ulong first, ulong second) => first < second;
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override ulong Increment(ulong value) => ++value;
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected override ulong Decrement(ulong value) => --value;
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override ulong Add(ulong first, ulong second) => first + second;
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected override ulong Subtract(ulong first, ulong second) => first - second;
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
67     {
68         ref var firstLink = ref Links[first];
69         ref var secondLink = ref Links[second];
70         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
71     }
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
75     {
76         ref var firstLink = ref Links[first];
77         ref var secondLink = ref Links[second];
78         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↪ secondLink.Source, secondLink.Target);
79     }
80
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
83
84     [MethodImpl(MethodImplOptions.AggressiveInlining)]
85     protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
86 }
87 }

```

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

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.United.Specific
6  {
7      public unsafe class UInt64LinksSourcesAvlBalancedTreeMethods :
    ↪ UInt64LinksAvlBalancedTreeMethodsBase
8      {
9          public UInt64LinksSourcesAvlBalancedTreeMethods(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].LeftAsSource;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetRightReference(ulong node) => ref
    ↪ Links[node].RightAsSource;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
22
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     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsSource);
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
33         ↳ Links[node].SizeAsSource, size);
34
35     [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 }
92

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Memory.United.Specific
6 {
7     public unsafe class UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods :
8         ↳ UInt64LinksRecursionlessSizeBalancedTreeMethodsBase
9     {
10

```

```

9      public UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods(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].LeftAsSource;

13
14      [MethodImpl(MethodImplOptions.AggressiveInlining)]
15      protected override ref ulong GetRightReference(ulong node) => ref
    ↪ Links[node].RightAsSource;

16
17      [MethodImpl(MethodImplOptions.AggressiveInlining)]
18      protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;

19
20      [MethodImpl(MethodImplOptions.AggressiveInlining)]
21      protected override ulong GetRight(ulong node) => Links[node].RightAsSource;

22
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 }

```

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

```

1      using System.Runtime.CompilerServices;
2
3      #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5      namespace Platform.Data.Doublets.Memory.United.Specific
6      {
7          public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
    ↪ UInt64LinksSizeBalancedTreeMethodsBase
8          {
9              public UInt64LinksSourcesSizeBalancedTreeMethods(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].LeftAsSource;

```

```

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

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Memory.United.Specific
6 {
7     public unsafe class UInt64LinksTargetsAvlBalancedTreeMethods :
8     ↪ UInt64LinksAvlBalancedTreeMethodsBase
9     {
10         public UInt64LinksTargetsAvlBalancedTreeMethods(LinksConstants<ulong> constants,
11         ↪ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
12         ↪ { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetLeftReference(ulong node) => ref
16         ↪ Links[node].LeftAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20         ↪ Links[node].RightAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
30         ↪ left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
34         ↪ right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override ulong GetSize(ulong node) => 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         => firstSource < secondSource || (firstSource == secondSource && firstTarget <
53         ↪ secondTarget);
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
57         ↪ ulong secondSource, ulong secondTarget)
58         => firstSource > secondSource || (firstSource == secondSource && firstTarget >
59         ↪ secondTarget);
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) => GetSizeValue(Links[node].SizeAsTarget);
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
35         ↳ Links[node].SizeAsTarget, size);
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected override bool GetLeftIsChild(ulong node) =>
39         ↳ GetLeftIsChildValue(Links[node].SizeAsTarget);
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override void SetLeftIsChild(ulong node, bool value) =>
43         ↳ SetLeftIsChildValue(ref Links[node].SizeAsTarget, value);
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override bool GetRightIsChild(ulong node) =>
47         ↳ GetRightIsChildValue(Links[node].SizeAsTarget);
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override void SetRightIsChild(ulong node, bool value) =>
51         ↳ SetRightIsChildValue(ref Links[node].SizeAsTarget, value);
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override sbyte GetBalance(ulong node) =>
55         ↳ GetBalanceValue(Links[node].SizeAsTarget);
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
59         ↳ Links[node].SizeAsTarget, value);
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     protected override ulong GetTreeRoot() => Header->RootAsTarget;
63
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
66
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
69         ↳ ulong secondSource, ulong secondTarget)
70         => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
71         ↳ secondSource);
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
75         ↳ ulong secondSource, ulong secondTarget)
76         => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
77         ↳ secondSource);
78
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     protected override void ClearNode(ulong node)
81     {
82         ref var link = ref Links[node];
83         link.LeftAsTarget = OUL;
84         link.RightAsTarget = OUL;
85         link.SizeAsTarget = OUL;
86     }
87 }
88 }

```

1.107 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsRecursionlessSizeBalancedTr

```

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

```

```

7 public unsafe class UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods :
  ↳ UInt64LinksRecursionlessSizeBalancedTreeMethodsBase
8 {
9     public UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods(LinksConstants<ulong>
  ↳ constants, RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants,
  ↳ links, header) { }

10
11     [MethodImpl(MethodImplOptions.AggressiveInlining)]
12     protected override ref ulong GetLeftReference(ulong node) => ref
  ↳ Links[node].LeftAsTarget;

13
14     [MethodImpl(MethodImplOptions.AggressiveInlining)]
15     protected override ref ulong GetRightReference(ulong node) => ref
  ↳ Links[node].RightAsTarget;

16
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;

19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;

22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
  ↳ left;

25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
  ↳ right;

28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;

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

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

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

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

44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
  ↳ ulong secondSource, ulong secondTarget)
47     => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
  ↳ secondSource);

48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override void ClearNode(ulong node)
51     {
52         ref var link = ref Links[node];
53         link.LeftAsTarget = OUL;
54         link.RightAsTarget = OUL;
55         link.SizeAsTarget = OUL;
56     }
57 }
58 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Memory.United.Specific
6 {
7     public unsafe class UInt64LinksTargetsSizeBalancedTreeMethods :
  ↳ UInt64LinksSizeBalancedTreeMethodsBase
8     {
9         public UInt64LinksTargetsSizeBalancedTreeMethods(LinksConstants<ulong> constants,
  ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
  ↳ { }
10

```

```

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

```

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

```

1 using System;
2 using System.Runtime.CompilerServices;
3 using Platform.Memory;
4 using Platform.Singletons;
5 using Platform.Data.Doublets.Memory.United.Generic;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Memory.United.Specific
10 {
11     /// <summary>
12     /// <para>Represents a low-level implementation of direct access to resizable memory, for
    ↳ organizing the storage of links with addresses represented as <see cref="ulong"
    ↳ />.</para>
13     /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
    ↳ размером, для организации хранения связей с адресами представленными в виде <see
    ↳ cref="ulong"/>.</para>
14     /// </summary>
15     public unsafe class UInt64UnitedMemoryLinks : UnitedMemoryLinksBase<ulong>
16     {

```



```

17 private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
18 private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
19 private LinksHeader<ulong>* _header;
20 private RawLink<ulong>* _links;
21
22 [MethodImpl(MethodImplOptions.AggressiveInlining)]
23 public UInt64UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
24
25 /// <summary>
26 /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
27   ↳ минимальным шагом расширения базы данных.
28 /// </summary>
29 /// <param name="address">Полный путь к файлу базы данных.</param>
30 /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
31   ↳ байтах.</param>
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 public UInt64UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
34   ↳ FileMappedResizableDirectMemory(address, memoryReservationStep),
35   ↳ memoryReservationStep) { }
36
37 [MethodImpl(MethodImplOptions.AggressiveInlining)]
38 public UInt64UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
39   ↳ DefaultLinksSizeStep) { }
40
41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
43   ↳ memoryReservationStep) : this(memory, memoryReservationStep,
44   ↳ Default<LinksConstants<ulong>>.Instance, IndexTreeType.Default) { }
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
48   ↳ memoryReservationStep, LinksConstants<ulong> constants, IndexTreeType indexTreeType)
49   ↳ : base(memory, memoryReservationStep, constants)
50 {
51     if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
52     {
53         _createSourceTreeMethods = () => new
54           ↳ UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
55         _createTargetTreeMethods = () => new
56           ↳ UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
57     }
58     else if (indexTreeType == IndexTreeType.SizeBalancedTree)
59     {
60         _createSourceTreeMethods = () => new
61           ↳ UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
62         _createTargetTreeMethods = () => new
63           ↳ UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
64     }
65     else
66     {
67         _createSourceTreeMethods = () => new
68           ↳ UInt64LinksSourcesRecursionlessSizeBalancedTreeMethods(Constants, _links,
69           ↳ _header);
70         _createTargetTreeMethods = () => new
71           ↳ UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods(Constants, _links,
72           ↳ _header);
73     }
74     Init(memory, memoryReservationStep);
75 }
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected override void SetPointers(IResizableDirectMemory memory)
79 {
80     _header = (LinksHeader<ulong>*)memory.Pointer;
81     _links = (RawLink<ulong>*)memory.Pointer;
82     SourcesTreeMethods = _createSourceTreeMethods();
83     TargetsTreeMethods = _createTargetTreeMethods();
84     UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
85 }
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 protected override void ResetPointers()
89 {
90     base.ResetPointers();
91     _links = null;
92     _header = null;
93 }

```

```

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
83     ↪ _links[linkIndex];
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     protected override bool AreEqual(ulong first, ulong second) => first == second;
87
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     protected override bool LessThan(ulong first, ulong second) => first < second;
90
91     [MethodImpl(MethodImplOptions.AggressiveInlining)]
92     protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
93
94     [MethodImpl(MethodImplOptions.AggressiveInlining)]
95     protected override bool GreaterThan(ulong first, ulong second) => first > second;
96
97     [MethodImpl(MethodImplOptions.AggressiveInlining)]
98     protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
99
100    [MethodImpl(MethodImplOptions.AggressiveInlining)]
101    protected override ulong GetZero() => 0UL;
102
103    [MethodImpl(MethodImplOptions.AggressiveInlining)]
104    protected override ulong GetOne() => 1UL;
105
106    [MethodImpl(MethodImplOptions.AggressiveInlining)]
107    protected override long ConvertToInt64(ulong value) => (long)value;
108
109    [MethodImpl(MethodImplOptions.AggressiveInlining)]
110    protected override ulong ConvertToAddress(long value) => (ulong)value;
111
112    [MethodImpl(MethodImplOptions.AggressiveInlining)]
113    protected override ulong Add(ulong first, ulong second) => first + second;
114
115    [MethodImpl(MethodImplOptions.AggressiveInlining)]
116    protected override ulong Subtract(ulong first, ulong second) => first - second;
117
118    [MethodImpl(MethodImplOptions.AggressiveInlining)]
119    protected override ulong Increment(ulong link) => ++link;
120
121    [MethodImpl(MethodImplOptions.AggressiveInlining)]
122    protected override ulong Decrement(ulong link) => --link;
123 }

```

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

```

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

```

1.111 ./csharp/Platform.Data.Doublets/Numbers/Rational/DecimalToRationalConverter.cs

```

1  using System.Numerics;
2  using Platform.Converters;

```

```

3 using Platform.Data.Doublets.Decorators;
4 using System.Globalization;
5 using Platform.Data.Doublets.Numbers.Raw;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Numbers.Rational
10 {
11     public class DecimalToRationalConverter<TLink> : LinksDecoratorBase<TLink>,
12         ↳ IConverter<decimal, TLink>
13         where TLink: struct
14     {
15         public readonly BigIntegerToRawNumberSequenceConverter<TLink>
16             ↳ BigIntegerToRawNumberSequenceConverter;
17
18         public DecimalToRationalConverter(ILinks<TLink> links,
19             ↳ BigIntegerToRawNumberSequenceConverter<TLink>
20             ↳ BigIntegerToRawNumberSequenceConverter) : base(links)
21         {
22             BigIntegerToRawNumberSequenceConverter = BigIntegerToRawNumberSequenceConverter;
23         }
24
25         public TLink Convert(decimal @decimal)
26         {
27             var decimalAsString = @decimal.ToString(CultureInfo.InvariantCulture);
28             var dotPosition = decimalAsString.IndexOf('.');
29             var decimalWithoutDots = decimalAsString;
30             int digitsAfterDot = 0;
31             if (dotPosition != -1)
32             {
33                 decimalWithoutDots = decimalWithoutDots.Remove(dotPosition, 1);
34                 digitsAfterDot = decimalAsString.Length - 1 - dotPosition;
35             }
36             BigInteger denominator = new(System.Math.Pow(10, digitsAfterDot));
37             BigInteger numerator = BigInteger.Parse(decimalWithoutDots);
38             BigInteger greatestCommonDivisor;
39             do
40             {
41                 greatestCommonDivisor = BigInteger.GreatestCommonDivisor(numerator, denominator);
42                 numerator /= greatestCommonDivisor;
43                 denominator /= greatestCommonDivisor;
44             }
45             while (greatestCommonDivisor > 1);
46             var numeratorLink = BigIntegerToRawNumberSequenceConverter.Convert(numerator);
47             var denominatorLink = BigIntegerToRawNumberSequenceConverter.Convert(denominator);
48             return _links.GetOrCreate(numeratorLink, denominatorLink);
49         }
50     }
51 }

```

1.112 ./csharp/Platform.Data.Doublets/Numbers/Rational/RationalToDecimalConverter.cs

```

1 using Platform.Converters;
2 using Platform.Data.Doublets.Decorators;
3 using Platform.Data.Doublets.Numbers.Raw;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Numbers.Rational
8 {
9     public class RationalToDecimalConverter<TLink> : LinksDecoratorBase<TLink>,
10         ↳ IConverter<TLink, decimal>
11         where TLink: struct
12     {
13         public readonly RawNumberSequenceToBigIntegerConverter<TLink>
14             ↳ RawNumberSequenceToBigIntegerConverter;
15
16         public RationalToDecimalConverter(ILinks<TLink> links,
17             ↳ RawNumberSequenceToBigIntegerConverter<TLink>
18             ↳ RawNumberSequenceToBigIntegerConverter) : base(links)
19         {
20             RawNumberSequenceToBigIntegerConverter = RawNumberSequenceToBigIntegerConverter;
21         }
22
23         public decimal Convert(TLink rationalNumber)
24         {
25             var numerator = (decimal)RawNumberSequenceToBigIntegerConverter.Convert(_links.GetSo
26                 ↳ urce(rationalNumber));
27             var denominator = (decimal)RawNumberSequenceToBigIntegerConverter.Convert(_links.Get
28                 ↳ Target(rationalNumber));
29             return numerator / denominator;
30         }
31     }
32 }

```

```

24     }
25 }
26 }

```

1.113 ./csharp/Platform.Data.Doublets/Numbers/Raw/BigIntegerToRawNumberSequenceConverter.cs

```

1  using System.Collections.Generic;
2  using System.Numerics;
3  using Platform.Converters;
4  using Platform.Data.Doublets.Decorators;
5  using Platform.Numbers;
6  using Platform.Reflection;
7  using Platform.Unsafe;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Numbers.Raw
12 {
13     public class BigIntegerToRawNumberSequenceConverter<TLink> : LinksDecoratorBase<TLink>,
14         ↪ IConverter<BigInteger, TLink>
15         where TLink : struct
16     {
17         public static readonly TLink MaximumValue = NumericType<TLink>.MaxValue;
18         public static readonly TLink BitMask = Bit.ShiftRight(MaximumValue, 1);
19         public readonly IConverter<TLink> AddressToNumberConverter;
20         public readonly IConverter<IList<TLink>, TLink> ListToSequenceConverter;
21         public readonly TLink NegativeNumberMarker;
22
23         public BigIntegerToRawNumberSequenceConverter(ILinks<TLink> links, IConverter<TLink>
24             ↪ addressToNumberConverter, IConverter<IList<TLink>, TLink> listToSequenceConverter,
25             ↪ TLink negativeNumberMarker) : base(links)
26         {
27             AddressToNumberConverter = addressToNumberConverter;
28             ListToSequenceConverter = listToSequenceConverter;
29             NegativeNumberMarker = negativeNumberMarker;
30         }
31
32         private List<TLink> GetRawNumberParts(BigInteger bigInteger)
33         {
34             List<TLink> rawNumbers = new();
35             BigInteger currentBigInt = bigInteger;
36             do
37             {
38                 var bigIntBytes = currentBigInt.ToByteArray();
39                 var bigIntWithBitMask = Bit.And(bigIntBytes.ToStructure<TLink>(), BitMask);
40                 var rawNumber = AddressToNumberConverter.Convert(bigIntWithBitMask);
41                 rawNumbers.Add(rawNumber);
42                 currentBigInt >>= 63;
43             }
44             while (currentBigInt > 0);
45             return rawNumbers;
46         }
47
48         public TLink Convert(BigInteger bigInteger)
49         {
50             var sign = bigInteger.Sign;
51             var number = GetRawNumberParts(sign == -1 ? BigInteger.Negate(bigInteger) :
52                 ↪ bigInteger);
53             var numberSequence = ListToSequenceConverter.Convert(number);
54             return sign == -1 ? _links.GetOrCreate(NegativeNumberMarker, numberSequence) :
55                 ↪ numberSequence;
56         }
57     }
58 }

```

1.114 ./csharp/Platform.Data.Doublets/Numbers/Raw/LongRawNumberSequenceToNumberConverter.cs

```

1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Stacks;
3  using Platform.Converters;
4  using Platform.Numbers;
5  using Platform.Reflection;
6  using Platform.Data.Doublets.Decorators;
7  using Platform.Data.Doublets.Sequences.Walkers;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Numbers.Raw
12 {
13     public class LongRawNumberSequenceToNumberConverter<TSource, TTarget> :
14         ↪ LinksDecoratorBase<TSource>, IConverter<TSource, TTarget>
15     {
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

```

15     private static readonly int _bitsPerRawNumber = NumericType<TSource>.BitsSize - 1;
16     private static readonly UncheckedConverter<TSource, TTarget> _sourceToTargetConverter =
    ↪ UncheckedConverter<TSource, TTarget>.Default;
17
18     private readonly IConverter<TSource> _numberToAddressConverter;
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     public LongRawNumberSequenceToNumberConverter(ILinks<TSource> links, IConverter<TSource>
    ↪ numberToAddressConverter) : base(links) => _numberToAddressConverter =
    ↪ numberToAddressConverter;
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     public TTarget Convert(TSource source)
25     {
26         var constants = Links.Constants;
27         var externalReferencesRange = constants.ExternalReferencesRange;
28         if (externalReferencesRange.HasValue &&
    ↪ externalReferencesRange.Value.Contains(source))
29         {
30             return
    ↪ _sourceToTargetConverter.Convert(_numberToAddressConverter.Convert(source));
31         }
32         else
33         {
34             var pair = Links.GetLink(source);
35             var walker = new LeftSequenceWalker<TSource>(Links, new DefaultStack<TSource>(),
    ↪ (link) => externalReferencesRange.HasValue &&
    ↪ externalReferencesRange.Value.Contains(link));
36             TTarget result = default;
37             foreach (var element in walker.Walk(source))
38             {
39                 result = Bit.Or(Bit.ShiftLeft(result, _bitsPerRawNumber), Convert(element));
40             }
41             return result;
42         }
43     }
44 }
45 }

```

1.115 ./csharp/Platform.Data.Doublets/Numbers/Raw/NumberToLongRawNumberSequenceConverter.cs

```

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

```

```

33         return Links.GetOrCreate(convertedNumber, Convert(Bit.ShiftRight(source,
    ↪      _bitsPerRowNumber)));
34     }
35     else
36     {
37         return
    ↪      _addressToNumberConverter.Convert(_sourceToTargetConverter.Convert(source));
38     }
39 }
40 }
41 }

```

1.116 ./csharp/Platform.Data.Doublets/Numbers/Row/RowNumberSequenceToBigIntegerConverter.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Numerics;
4  using Platform.Collections.Stacks;
5  using Platform.Converters;
6  using Platform.Data.Doublets.Decorators;
7  using Platform.Data.Doublets.Sequences.Walkers;
8  using Platform.Unsafe;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.Numbers.Row
13 {
14     public class RowNumberSequenceToBigIntegerConverter<TLink> : LinksDecoratorBase<TLink>,
    ↪      IConverter<TLink, BigInteger>
    where TLink : struct
15     {
16     {
17         public readonly EqualityComparer<TLink> EqualityComparer =
    ↪      EqualityComparer<TLink>.Default;
18         public readonly IConverter<TLink, TLink> NumberToAddressConverter;
19         public readonly LeftSequenceWalker<TLink> LeftSequenceWalker;
20         public readonly TLink NegativeNumberMarker;
21
22         public RowNumberSequenceToBigIntegerConverter(ILinks<TLink> links, IConverter<TLink,
    ↪      TLink> numberToAddressConverter, TLink negativeNumberMarker) : base(links)
23         {
24             NumberToAddressConverter = numberToAddressConverter;
25             LeftSequenceWalker = new(links, new DefaultStack<TLink>());
26             NegativeNumberMarker = negativeNumberMarker;
27         }
28
29         public BigInteger Convert(TLink bigInteger)
30         {
31             var sign = 1;
32             var bigIntegerSequence = bigInteger;
33             if (EqualityComparer.Equals(_links.GetSource(bigIntegerSequence),
    ↪      NegativeNumberMarker))
34             {
35                 sign = -1;
36                 bigIntegerSequence = _links.GetTarget(bigInteger);
37             }
38             using var enumerator = LeftSequenceWalker.Walk(bigIntegerSequence).GetEnumerator();
39             if (!enumerator.MoveNext())
40             {
41                 throw new Exception("Raw number sequence cannot be empty.");
42             }
43             var nextPart = NumberToAddressConverter.Convert(enumerator.Current);
44             BigInteger currentBigInt = new(nextPart.ToBytes());
45             while (enumerator.MoveNext())
46             {
47                 currentBigInt <= 63;
48                 nextPart = NumberToAddressConverter.Convert(enumerator.Current);
49                 currentBigInt |= new BigInteger(nextPart.ToBytes());
50             }
51             return sign == -1 ? BigInteger.Negate(currentBigInt) : currentBigInt;
52         }
53     }
54 }

```

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

```

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

```

```

8
9 namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
12         ↳ IConverter<TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↳ EqualityComparer<TLink>.Default;
16         private static readonly TLink _zero = default;
17         private static readonly TLink _one = Arithmetic.Increment(_zero);
18
19         private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
23             ↳ powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
24             ↳ powerOf2ToUnaryNumberConverter;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Convert(TLink number)
28         {
29             var links = _links;
30             var nullConstant = links.Constants.Null;
31             var target = nullConstant;
32             for (var i = 0; !_equalityComparer.Equals(number, _zero) && i <
33                 ↳ NumericType<TLink>.BitsSize; i++)
34             {
35                 if (_equalityComparer.Equals(Bit.And(number, _one), _one))
36                 {
37                     target = _equalityComparer.Equals(target, nullConstant)
38                         ? _powerOf2ToUnaryNumberConverter.Convert(i)
39                         : links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
40                 }
41                 number = Bit.ShiftRight(number, 1);
42             }
43             return target;
44         }
45     }
46 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using Platform.Interfaces;
4 using Platform.Converters;
5 using System.Runtime.CompilerServices;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
12         ↳ IConverter<Doublet<TLink>, TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↳ EqualityComparer<TLink>.Default;
16
17         private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
18         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public LinkToItsFrequencyNumberConveter(
22             ILinks<TLink> links,
23             IProperty<TLink, TLink> frequencyPropertyOperator,
24             IConverter<TLink> unaryNumberToAddressConverter)
25             : base(links)
26         {
27             _frequencyPropertyOperator = frequencyPropertyOperator;
28             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public TLink Convert(Doublet<TLink> doublet)
33         {
34             var links = _links;
35             var link = links.SearchOrDefault(doublet.Source, doublet.Target);
36             if (_equalityComparer.Equals(link, default))
37             {
38                 throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
39             }
40         }
41     }
42 }

```

```

38     var frequency = _frequencyPropertyOperator.Get(link);
39     if (_equalityComparer.Equals(frequency, default))
40     {
41         return default;
42     }
43     var frequencyNumber = links.GetSource(frequency);
44     return _unaryNumberToAddressConverter.Convert(frequencyNumber);
45 }
46 }
47 }

```

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

```

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

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Converters;
4  using Platform.Numbers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Numbers.Unary
9  {
10     public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>,
11         ⇨ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ⇨ EqualityComparer<TLink>.Default;
15         private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
16             ⇨ UncheckedConverter<TLink, ulong>.Default;
17         private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
18             ⇨ UncheckedConverter<ulong, TLink>.Default;
19         private static readonly TLink _zero = default;
20         private static readonly TLink _one = Arithmetic.Increment(_zero);
21
22         private readonly Dictionary<TLink, TLink> _unaryToUInt64;
23         private readonly TLink _unaryOne;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

22     public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
23         : base(links)
24     {
25         _unaryOne = unaryOne;
26         _unaryToUInt64 = CreateUnaryToUInt64Dictionary(links, unaryOne);
27     }
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public TLink Convert(TLink unaryNumber)
31     {
32         if (_equalityComparer.Equals(unaryNumber, default))
33         {
34             return default;
35         }
36         if (_equalityComparer.Equals(unaryNumber, _unaryOne))
37         {
38             return _one;
39         }
40         var links = _links;
41         var source = links.GetSource(unaryNumber);
42         var target = links.GetTarget(unaryNumber);
43         if (_equalityComparer.Equals(source, target))
44         {
45             return _unaryToUInt64[unaryNumber];
46         }
47         else
48         {
49             var result = _unaryToUInt64[source];
50             TLink lastValue;
51             while (!_unaryToUInt64.TryGetValue(target, out lastValue))
52             {
53                 source = links.GetSource(target);
54                 result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
55                 target = links.GetTarget(target);
56             }
57             result = Arithmetic<TLink>.Add(result, lastValue);
58             return result;
59         }
60     }
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     private static Dictionary<TLink, TLink> CreateUnaryToUInt64Dictionary(ILinks<TLink>
64     ↪ links, TLink unaryOne)
65     {
66         var unaryToUInt64 = new Dictionary<TLink, TLink>
67         {
68             { unaryOne, _one }
69         };
70         var unary = unaryOne;
71         var number = _one;
72         for (var i = 1; i < 64; i++)
73         {
74             unary = links.GetOrCreate(unary, unary);
75             number = Double(number);
76             unaryToUInt64.Add(unary, number);
77         }
78         return unaryToUInt64;
79     }
80
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     private static TLink Double(TLink number) =>
83     ↪ _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Reflection;
4  using Platform.Converters;
5  using Platform.Numbers;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
12     ↪ IConverter<TLink>
13     {

```

```

13     private static readonly EqualityComparer<TLink> _equalityComparer =
14         ↳ EqualityComparer<TLink>.Default;
15     private static readonly TLink _zero = default;
16     private static readonly TLink _one = Arithmetic.Increment(_zero);
17
18     private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
22         ↳ TLink> powerOf2ToUnaryNumberConverter) : base(links) => _unaryNumberPowerOf2Indicies
23         ↳ = CreateUnaryNumberPowerOf2IndiciesDictionary(powerOf2ToUnaryNumberConverter);
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     public TLink Convert(TLink sourceNumber)
27     {
28         var links = _links;
29         var nullConstant = links.Constants.Null;
30         var source = sourceNumber;
31         var target = nullConstant;
32         if (!_equalityComparer.Equals(source, nullConstant))
33         {
34             while (true)
35             {
36                 if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
37                 {
38                     SetBit(ref target, powerOf2Index);
39                     break;
40                 }
41                 else
42                 {
43                     powerOf2Index = _unaryNumberPowerOf2Indicies[links.GetSource(source)];
44                     SetBit(ref target, powerOf2Index);
45                     source = links.GetTarget(source);
46                 }
47             }
48         }
49         return target;
50     }
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     private static Dictionary<TLink, int>
54         ↳ CreateUnaryNumberPowerOf2IndiciesDictionary(IConverter<int, TLink>
55         ↳ powerOf2ToUnaryNumberConverter)
56     {
57         var unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
58         for (int i = 0; i < NumericType<TLink>.BitsSize; i++)
59         {
60             unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
61         }
62         return unaryNumberPowerOf2Indicies;
63     }
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     private static void SetBit(ref TLink target, int powerOf2Index) => target =
67         ↳ Bit.Or(target, Bit.ShiftLeft(_one, powerOf2Index));
68 }
69
70 }

```

1.122 ./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;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public PropertiesOperator(ILinks<TLink> links) : base(links) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public TLink GetValue(TLink @object, TLink property)
20         {
21             var links = _links;

```

```

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

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.PropertyOperators
8  {
9      public class PropertyOperator<TLink> : LinksOperatorBase<TLink>, IProperty<TLink, TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↳ EqualityComparer<TLink>.Default;
13
14         private readonly TLink _propertyMarker;
15         private readonly TLink _propertyValueMarker;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
19             ↳ propertyValueMarker) : base(links)
20         {
21             _propertyMarker = propertyMarker;
22             _propertyValueMarker = propertyValueMarker;
23         }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public TLink Get(TLink link)
27         {
28             var property = _links.SearchOrDefault(link, _propertyMarker);
29             return GetValue(GetContainer(property));
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         private TLink GetContainer(TLink property)
34         {
35             var valueContainer = default(TLink);
36             if (_equalityComparer.Equals(property, default))
37             {
38                 return valueContainer;
39             }
40             var links = _links;
41             var constants = links.Constants;
42             var countinueConstant = constants.Continue;
43             var breakConstant = constants.Break;
44             var anyConstant = constants.Any;
45             var query = new Link<TLink>(anyConstant, property, anyConstant);
46             links.Each(candidate =>
47             {
48                 var candidateTarget = links.GetTarget(candidate);
49                 var valueTarget = links.GetTarget(candidateTarget);
50                 if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
51                 {
52                     valueContainer = links.GetIndex(candidate);
53                 }
54             });
55         }
56     }
57 }

```

```

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 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Converters
7  {
8      public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public override TLink Convert(ICollection<TLink> sequence)
15         {
16             var length = sequence.Count;
17             if (length < 1)
18             {
19                 return default;
20             }
21             if (length == 1)
22             {
23                 return sequence[0];
24             }
25             // Make copy of next layer
26             if (length > 2)
27             {
28                 // TODO: Try to use stackalloc (which at the moment is not working with
29                 ↪ generics) but will be possible with Sigil
30                 var halvedSequence = new TLink[(length / 2) + (length % 2)];
31                 HalveSequence(halvedSequence, sequence, length);
32                 sequence = halvedSequence;
33                 length = halvedSequence.Length;
34             }
35             // Keep creating layer after layer
36             while (length > 2)
37             {
38                 HalveSequence(sequence, sequence, length);
39                 length = (length / 2) + (length % 2);
40             }
41             return _links.GetOrCreate(sequence[0], sequence[1]);
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         private void HalveSequence(ICollection<TLink> destination, ICollection<TLink> source, int length)
45         {
46             var loopedLength = length - (length % 2);
47             for (var i = 0; i < loopedLength; i += 2)
48             {

```

```

49         destination[i / 2] = _links.GetOrCreate(source[i], source[i + 1]);
50     }
51     if (length > loopedLength)
52     {
53         destination[length / 2] = source[length - 1];
54     }
55 }
56 }
57 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections;
5  using Platform.Converters;
6  using Platform.Singletons;
7  using Platform.Numbers;
8  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.Sequences.Converters
13 {
14     /// <remarks>
15     /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от
16     ///     Links на этапе сжатия.
17     ///     А именно будет создаваться временный список пар необходимых для выполнения сжатия, в
18     ///     таком случае тип значения элемента массива может быть любым, как char так и ulong.
19     ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
20     ///     пар, а так же разом выполнить замену.
21     /// </remarks>
22     public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
23     {
24         private static readonly LinksConstants<TLink> _constants =
25             ↪ Default<LinksConstants<TLink>>.Instance;
26         private static readonly EqualityComparer<TLink> _equalityComparer =
27             ↪ EqualityComparer<TLink>.Default;
28         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
29
30         private static readonly TLink _zero = default;
31         private static readonly TLink _one = Arithmetic.Increment(_zero);
32
33         private readonly IConverter<IList<TLink>, TLink> _baseConverter;
34         private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
35         private readonly TLink _minFrequencyToCompress;
36         private readonly bool _doInitialFrequenciesIncrement;
37         private Doublet<TLink> _maxDoublet;
38         private LinkFrequency<TLink> _maxDoubletData;
39
40         private struct HalfDoublet
41         {
42             public TLink Element;
43             public LinkFrequency<TLink> DoubletData;
44
45             [MethodImpl(MethodImplOptions.AggressiveInlining)]
46             public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
47             {
48                 Element = element;
49                 DoubletData = doubletData;
50             }
51
52             public override string ToString() => $"{Element}: ({DoubletData})";
53         }
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
57             ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
58             : this(links, baseConverter, doubletFrequenciesCache, _one, true) { }
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
62             ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
63             ↪ doInitialFrequenciesIncrement)
64             : this(links, baseConverter, doubletFrequenciesCache, _one,
65                 ↪ doInitialFrequenciesIncrement) { }
66
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
69             ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
70             ↪ minFrequencyToCompress, bool doInitialFrequenciesIncrement)

```

```

60         : base(links)
61     {
62         _baseConverter = baseConverter;
63         _doubletFrequenciesCache = doubletFrequenciesCache;
64         if (_comparer.Compare(minFrequencyToCompress, _one) < 0)
65         {
66             minFrequencyToCompress = _one;
67         }
68         _minFrequencyToCompress = minFrequencyToCompress;
69         _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
70         ResetMaxDoublet();
71     }
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     public override TLink Convert(ICollection<TLink> source) =>
75     {
76         // <remarks>
77         // Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
78         // Faster version (doublets' frequencies dictionary is not recreated).
79         // </remarks>
80         [MethodImpl(MethodImplOptions.AggressiveInlining)]
81         private ICollection<TLink> Compress(ICollection<TLink> sequence)
82         {
83             if (sequence.IsNullOrEmpty())
84             {
85                 return null;
86             }
87             if (sequence.Count == 1)
88             {
89                 return sequence;
90             }
91             if (sequence.Count == 2)
92             {
93                 return new[] { _links.GetOrCreate(sequence[0], sequence[1]) };
94             }
95             // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
96             var copy = new HalfDoublet[sequence.Count];
97             Doublet<TLink> doublet = default;
98             for (var i = 1; i < sequence.Count; i++)
99             {
100                 doublet = new Doublet<TLink>(sequence[i - 1], sequence[i]);
101                 LinkFrequency<TLink> data;
102                 if (_doInitialFrequenciesIncrement)
103                 {
104                     data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
105                 }
106                 else
107                 {
108                     data = _doubletFrequenciesCache.GetFrequency(ref doublet);
109                     if (data == null)
110                     {
111                         throw new NotSupportedException("If you ask not to increment
112                             ↪ frequencies, it is expected that all frequencies for the sequence
113                             ↪ are prepared.");
114                     }
115                 }
116                 copy[i - 1].Element = sequence[i - 1];
117                 copy[i - 1].DoubletData = data;
118                 UpdateMaxDoublet(ref doublet, data);
119             }
120             copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
121             copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
122             if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
123             {
124                 var newLength = ReplaceDoublets(copy);
125                 sequence = new TLink[newLength];
126                 for (int i = 0; i < newLength; i++)
127                 {
128                     sequence[i] = copy[i].Element;
129                 }
130             }
131             return sequence;
132         }
133     }
134
135     // <remarks>
136     // Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
137     // </remarks>
138     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

136 private int ReplaceDoublets(HalfDoublet[] copy)
137 {
138     var oldLength = copy.Length;
139     var newLength = copy.Length;
140     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
141     {
142         var maxDoubletSource = _maxDoublet.Source;
143         var maxDoubletTarget = _maxDoublet.Target;
144         if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
145         {
146             _maxDoubletData.Link = _links.GetOrCreate(maxDoubletSource,
147                 ↪ maxDoubletTarget);
148         }
149         var maxDoubletReplacementLink = _maxDoubletData.Link;
150         oldLength--;
151         var oldLengthMinusTwo = oldLength - 1;
152         // Substitute all usages
153         int w = 0, r = 0; // (r == read, w == write)
154         for (; r < oldLength; r++)
155         {
156             if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
157                 ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
158             {
159                 if (r > 0)
160                 {
161                     var previous = copy[w - 1].Element;
162                     copy[w - 1].DoubletData.DecrementFrequency();
163                     copy[w - 1].DoubletData =
164                         ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
165                             ↪ maxDoubletReplacementLink);
166                 }
167                 if (r < oldLengthMinusTwo)
168                 {
169                     var next = copy[r + 2].Element;
170                     copy[r + 1].DoubletData.DecrementFrequency();
171                     copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
172                         ↪ next);
173                 }
174                 copy[w++] = copy[r];
175                 newLength--;
176             }
177             else
178             {
179                 copy[w++] = copy[r];
180             }
181             oldLength = newLength;
182             ResetMaxDoublet();
183             UpdateMaxDoublet(copy, newLength);
184         }
185         return newLength;
186     }
187 }
188
189 [MethodImpl(MethodImplOptions.AggressiveInlining)]
190 private void ResetMaxDoublet()
191 {
192     _maxDoublet = new Doublet<TLink>();
193     _maxDoubletData = new LinkFrequency<TLink>();
194 }
195
196 [MethodImpl(MethodImplOptions.AggressiveInlining)]
197 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
198 {
199     Doublet<TLink> doublet = default;
200     for (var i = 1; i < length; i++)
201     {
202         doublet = new Doublet<TLink>(copy[i - 1].Element, copy[i].Element);
203         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
204     }
205 }
206
207 [MethodImpl(MethodImplOptions.AggressiveInlining)]
208 private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)

```

```

209     {
210         var frequency = data.Frequency;
211         var maxFrequency = _maxDoubletData.Frequency;
212         //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
213             ↳ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
214             ↳ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
215             ↳ _maxDoublet.Target)))
216         if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
217             ↳ (_comparer.Compare(maxFrequency, frequency) < 0 ||
218             ↳ (_equalityComparer.Equals(maxFrequency, frequency) &&
219             ↳ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
220             ↳ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
221             ↳ better stability and better compression on sequent data and even on random
222             ↳ numbers data (but gives collisions anyway) */
223         {
224             _maxDoublet = doublet;
225             _maxDoubletData = data;
226         }
227     }
228 }
229 }
230 }
231 }

```

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

```

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

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Collections.Lists;
4  using Platform.Converters;
5  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
6  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Sequences.Converters
11 {
12     public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↳ EqualityComparer<TLink>.Default;
16         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
17
18         private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
22             ↳ sequenceToItsLocalElementLevelsConverter) : base(links)
23             => _sequenceToItsLocalElementLevelsConverter =
24             ↳ sequenceToItsLocalElementLevelsConverter;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public OptimalVariantConverter(ILinks<TLink> links, LinkFrequenciesCache<TLink>
28             ↳ linkFrequenciesCache)
29             : this(links, new SequenceToItsLocalElementLevelsConverter<TLink>(links, new Frequen_
30                 ↳ ciesCacheBasedLinkToItsFrequencyNumberConverter<TLink>(linkFrequenciesCache))) { }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public OptimalVariantConverter(ILinks<TLink> links)
34             : this(links, new LinkFrequenciesCache<TLink>(links, new
35                 ↳ TotalSequenceSymbolFrequencyCounter<TLink>(links))) { }
36     }
37 }

```



```

30
31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 public override TLink Convert(ICollection<TLink> sequence)
33 {
34     var length = sequence.Count;
35     if (length == 1)
36     {
37         return sequence[0];
38     }
39     if (length == 2)
40     {
41         return _links.GetOrCreate(sequence[0], sequence[1]);
42     }
43     sequence = sequence.ToArray();
44     var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
45     while (length > 2)
46     {
47         var levelRepeat = 1;
48         var currentLevel = levels[0];
49         var previousLevel = levels[0];
50         var skipOnce = false;
51         var w = 0;
52         for (var i = 1; i < length; i++)
53         {
54             if (_equalityComparer.Equals(currentLevel, levels[i]))
55             {
56                 levelRepeat++;
57                 skipOnce = false;
58                 if (levelRepeat == 2)
59                 {
60                     sequence[w] = _links.GetOrCreate(sequence[i - 1], sequence[i]);
61                     var newLevel = i >= length - 1 ?
62                         GetPreviousLowerThanCurrentOrCurrent(previousLevel,
63                             ↪ currentLevel) :
64                         i < 2 ?
65                         GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
66                         GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
67                             ↪ currentLevel, levels[i + 1]);
68                     levels[w] = newLevel;
69                     previousLevel = currentLevel;
70                     w++;
71                     levelRepeat = 0;
72                     skipOnce = true;
73                 }
74                 else if (i == length - 1)
75                 {
76                     sequence[w] = sequence[i];
77                     levels[w] = levels[i];
78                     w++;
79                 }
80             }
81             else
82             {
83                 currentLevel = levels[i];
84                 levelRepeat = 1;
85                 if (skipOnce)
86                 {
87                     skipOnce = false;
88                 }
89                 else
90                 {
91                     sequence[w] = sequence[i - 1];
92                     levels[w] = levels[i - 1];
93                     previousLevel = levels[w];
94                     w++;
95                 }
96                 if (i == length - 1)
97                 {
98                     sequence[w] = sequence[i];
99                     levels[w] = levels[i];
100                     w++;
101                 }
102             }
103         }
104         length = w;
105     }
106     return _links.GetOrCreate(sequence[0], sequence[1]);
107 }

```

```

107     [MethodImpl(MethodImplOptions.AggressiveInlining)]
108     private static TLink GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink previous, TLink
    ↪ current, TLink next)
109     {
110         return _comparer.Compare(previous, next) > 0
111             ? _comparer.Compare(previous, current) < 0 ? previous : current
112             : _comparer.Compare(next, current) < 0 ? next : current;
113     }
114
115     [MethodImpl(MethodImplOptions.AggressiveInlining)]
116     private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
    ↪ _comparer.Compare(next, current) < 0 ? next : current;
117
118     [MethodImpl(MethodImplOptions.AggressiveInlining)]
119     private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
    ↪ => _comparer.Compare(previous, current) < 0 ? previous : current;
120 }
121 }

```

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

```

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

```

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

```

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

```

```

15     }
16 }

```

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

```

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

```

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

```

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

```

```

41         _stack.Push(source);
42         cursor = target;
43     }
44 }
45 var left = cursor;
46 var right = appendant;
47 while (!_equalityComparer.Equals(cursor = _stack.PopOrDefault(),
48     ↪ links.Constants.Null))
49 {
50     right = links.GetOrCreate(left, right);
51     left = cursor;
52 }
53 return links.GetOrCreate(left, right);
54 }
55 }

```

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

```

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

```

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

```

1 using System;
2 using System.Linq;
3 using System.Collections.Generic;
4 using System.Runtime.CompilerServices;
5 using Platform.Interfaces;
6 using Platform.Collections;
7 using Platform.Collections.Lists;
8 using Platform.Collections.Segments;
9 using Platform.Collections.Segments.Walkers;
10 using Platform.Singletons;
11 using Platform.Converters;
12 using Platform.Data.Doublets.Unicode;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     public class DuplicateSegmentsProvider<TLink> :
19         ↪ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
20         ↪ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
21     {
22         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
23             ↪ UncheckedConverter<TLink, long>.Default;
24         private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
25             ↪ UncheckedConverter<TLink, ulong>.Default;
26         private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
27             ↪ UncheckedConverter<ulong, TLink>.Default;
28
29         private readonly ILinks<TLink> _links;
30         private readonly ILinks<TLink> _sequences;
31         private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
32         private BitString _visited;
33
34         private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
35             ↪ IList<TLink>>>
36         {
37             private readonly IListEqualityComparer<TLink> _listComparer;
38         }
39     }
40 }

```

```

33     public ItemEquilityComparer() => _listComparer =
34         ↳ Default<IListEqualityComparer<TLink>>.Instance;
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
38         ↳ KeyValuePair<IList<TLink>, IList<TLink>> right) =>
39         ↳ _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
40         ↳ right.Value);
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
44         ↳ (_listComparer.GetHashCode(pair.Key),
45         ↳ _listComparer.GetHashCode(pair.Value)).GetHashCode();
46 }
47
48 private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
49 {
50     private readonly IListComparer<TLink> _listComparer;
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
57         ↳ KeyValuePair<IList<TLink>, IList<TLink>> right)
58     {
59         var intermediateResult = _listComparer.Compare(left.Key, right.Key);
60         if (intermediateResult == 0)
61         {
62             intermediateResult = _listComparer.Compare(left.Value, right.Value);
63         }
64         return intermediateResult;
65     }
66 }
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
70     : base(minimumStringSegmentLength: 2)
71 {
72     _links = links;
73     _sequences = sequences;
74 }
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
78 {
79     _groups = new HashSet<KeyValuePair<IList<TLink>,
80         ↳ IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
81     var links = _links;
82     var count = links.Count();
83     _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
84     links.Each(link =>
85     {
86         var linkIndex = links.GetIndex(link);
87         var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
88         var constants = links.Constants;
89         if (!_visited.Get(linkBitIndex))
90         {
91             var sequenceElements = new List<TLink>();
92             var filler = new ListFiller<TLink, TLink>(sequenceElements, constants.Break);
93             _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
94                 ↳ LinkAddress<TLink>(linkIndex));
95             if (sequenceElements.Count > 2)
96             {
97                 WalkAll(sequenceElements);
98             }
99         }
100         return constants.Continue;
101     });
102     var resultList = _groups.ToList();
103     var comparer = Default<ItemComparer>.Instance;
104     resultList.Sort(comparer);
105
106     #if DEBUG
107     foreach (var item in resultList)
108     {
109         PrintDuplicates(item);
110     }
111
112     #endif
113     return resultList;

```

```

103     }
104
105     [MethodImpl(MethodImplOptions.AggressiveInlining)]
106     protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
        ↪ length) => new Segment<TLink>(elements, offset, length);
107
108     [MethodImpl(MethodImplOptions.AggressiveInlining)]
109     protected override void OnDuplicateFound(Segment<TLink> segment)
110     {
111         var duplicates = CollectDuplicatesForSegment(segment);
112         if (duplicates.Count > 1)
113         {
114             _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
                ↪ duplicates));
115         }
116     }
117
118     [MethodImpl(MethodImplOptions.AggressiveInlining)]
119     private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
120     {
121         var duplicates = new List<TLink>();
122         var readAsElement = new HashSet<TLink>();
123         var restrictions = segment.ShiftRight();
124         var constants = _links.Constants;
125         restrictions[0] = constants.Any;
126         _sequences.Each(sequence =>
127         {
128             var sequenceIndex = sequence[constants.IndexPart];
129             duplicates.Add(sequenceIndex);
130             readAsElement.Add(sequenceIndex);
131             return constants.Continue;
132         }, restrictions);
133         if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
134         {
135             return new List<TLink>();
136         }
137         foreach (var duplicate in duplicates)
138         {
139             var duplicateBitIndex = _addressToInt64Converter.Convert(duplicate);
140             _visited.Set(duplicateBitIndex);
141         }
142         if (_sequences is Sequences sequencesExperiments)
143         {
144             var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
                ↪ ashSet<ulong>)(object)readAsElement,
                ↪ (IList<ulong>)segment);
145             foreach (var partiallyMatchedSequence in partiallyMatched)
146             {
147                 var sequenceIndex =
148                     ↪ _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
149                 duplicates.Add(sequenceIndex);
150             }
151         }
152         duplicates.Sort();
153         return duplicates;
154     }
155
156     [MethodImpl(MethodImplOptions.AggressiveInlining)]
157     private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
158     {
159         if (!(_links is ILinks<ulong> ulongLinks))
160         {
161             return;
162         }
163         var duplicatesKey = duplicatesItem.Key;
164         var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
165         Console.WriteLine($"{> {keyString} ({string.Join(", ", duplicatesKey)})");
166         var duplicatesList = duplicatesItem.Value;
167         for (int i = 0; i < duplicatesList.Count; i++)
168         {
169             var sequenceIndex = _addressToUInt64Converter.Convert(duplicatesList[i]);
170             var formattedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
                ↪ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
                ↪ UnicodeMap.IsCharLink(link.Index) ?
                ↪ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
171             Console.WriteLine(formattedSequenceStructure);
172             var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
                ↪ ulongLinks);

```

```

172         Console.WriteLine(sequenceString);
173     }
174     Console.WriteLine();
175 }
176 }
177 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5  using Platform.Numbers;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
10 {
11     /// <remarks>
12     /// Can be used to operate with many CompressingConverters (to keep global frequencies data
13     /// ↪ between them).
14     /// TODO: Extract interface to implement frequencies storage inside Links storage
15     /// </remarks>
16     public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
17     {
18         private static readonly EqualityComparer<TLink> _equalityComparer =
19             ↪ EqualityComparer<TLink>.Default;
20         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
21
22         private static readonly TLink _zero = default;
23         private static readonly TLink _one = Arithmetic.Increment(_zero);
24
25         private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
26         private readonly ICounter<TLink, TLink> _frequencyCounter;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
30             : base(links)
31         {
32             _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
33                 ↪ DoubletComparer<TLink>.Default);
34             _frequencyCounter = frequencyCounter;
35         }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
39         {
40             var doublet = new Doublet<TLink>(source, target);
41             return GetFrequency(ref doublet);
42         }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
46         {
47             _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
48             return data;
49         }
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         public void IncrementFrequencies(IList<TLink> sequence)
53         {
54             for (var i = 1; i < sequence.Count; i++)
55             {
56                 IncrementFrequency(sequence[i - 1], sequence[i]);
57             }
58         }
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
62         {
63             var doublet = new Doublet<TLink>(source, target);
64             return IncrementFrequency(ref doublet);
65         }
66
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         public void PrintFrequencies(IList<TLink> sequence)
69         {
70             for (var i = 1; i < sequence.Count; i++)
71             {
72                 PrintFrequency(sequence[i - 1], sequence[i]);
73             }
74         }
75     }
76 }

```

```

70     }
71 }
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 public void PrintFrequency(TLink source, TLink target)
75 {
76     var number = GetFrequency(source, target).Frequency;
77     Console.WriteLine("{0},{1} - {2}", source, target, number);
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
82 {
83     if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
84     {
85         data.IncrementFrequency();
86     }
87     else
88     {
89         var link = _links.SearchOrDefault(doublet.Source, doublet.Target);
90         data = new LinkFrequency<TLink>(_one, link);
91         if (!_equalityComparer.Equals(link, default))
92         {
93             data.Frequency = Arithmetic.Add(data.Frequency,
94                 ↪ _frequencyCounter.Count(link));
95         }
96         _doubletsCache.Add(doublet, data);
97     }
98     return data;
99 }
100
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 public void ValidateFrequencies()
103 {
104     foreach (var entry in _doubletsCache)
105     {
106         var value = entry.Value;
107         var linkIndex = value.Link;
108         if (!_equalityComparer.Equals(linkIndex, default))
109         {
110             var frequency = value.Frequency;
111             var count = _frequencyCounter.Count(linkIndex);
112             // TODO: Why `frequency` always greater than `count` by 1?
113             if (((_comparer.Compare(frequency, count) > 0) &&
114                 ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count), _one) > 0))
115                 || ((_comparer.Compare(count, frequency) > 0) &&
116                 ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency), _one) > 0)))
117             {
118                 throw new InvalidOperationException("Frequencies validation failed.");
119             }
120             //else
121             //{
122             //     if (value.Frequency > 0)
123             //     {
124             //         var frequency = value.Frequency;
125             //         linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
126             //         var count = _countLinkFrequency(linkIndex);
127             //         if ((frequency > count && frequency - count > 1) || (count > frequency
128             //             ↪ && count - frequency > 1))
129             //             throw new InvalidOperationException("Frequencies validation
130             //             ↪ failed.");
131             //     }
132             // }
133 }

```

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

```

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

```



```

8     public class LinkFrequency<TLink>
9     {
10         public TLink Frequency { get; set; }
11         public TLink Link { get; set; }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public LinkFrequency(TLink frequency, TLink link)
15         {
16             Frequency = frequency;
17             Link = link;
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public LinkFrequency() { }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public void IncrementFrequency() => Frequency = Arithmetic<TLink>.Increment(Frequency);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public override string ToString() => $"F: {Frequency}, L: {Link}";
31     }
32 }

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Interfaces;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7 {
8     public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
9     {
10         private readonly ILinks<TLink> _links;
11         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
15             ↪ ICriterionMatcher<TLink> markedSequenceMatcher)
16         {
17             _links = links;
18             _markedSequenceMatcher = markedSequenceMatcher;
19         }
20     }
21 }

```

```

19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     public TLink Count(TLink argument) => new
21     ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
22     ↪ _markedSequenceMatcher, argument).Count();
23 }

```

1.140 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequency

```

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

```

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

```

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

```

1.142 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOff

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Interfaces;
4 using Platform.Numbers;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
9 {
10     public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13         ↪ EqualityComparer<TLink>.Default;
14         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
15
16         protected readonly ILinks<TLink> _links;
17         protected readonly TLink _symbol;
18         protected readonly HashSet<TLink> _visits;
19     }
20 }

```

```

18     protected TLink _total;
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
22     {
23         _links = links;
24         _symbol = symbol;
25         _visits = new HashSet<TLink>();
26         _total = default;
27     }
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public TLink Count()
31     {
32         if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
33         {
34             return _total;
35         }
36         CountCore(_symbol);
37         return _total;
38     }
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     private void CountCore(TLink link)
42     {
43         var any = _links.Constants.Any;
44         if (_equalityComparer.Equals(_links.Count(any, link), default))
45         {
46             CountSequenceSymbolFrequency(link);
47         }
48         else
49         {
50             _links.Each(EachElementHandler, any, link);
51         }
52     }
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected virtual void CountSequenceSymbolFrequency(TLink link)
56     {
57         var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
58             ↪ link, _symbol);
59         _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
60     }
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     private TLink EachElementHandler(IList<TLink> doublet)
64     {
65         var constants = _links.Constants;
66         var doubletIndex = doublet[constants.IndexPart];
67         if (_visits.Add(doubletIndex))
68         {
69             CountCore(doubletIndex);
70         }
71         return constants.Continue;
72     }
73 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4  using Platform.Converters;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.HeightProviders
9  {
10     public class CachedSequenceHeightProvider<TLink> : ISequenceHeightProvider<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↪ EqualityComparer<TLink>.Default;
14
15         private readonly TLink _heightPropertyMarker;
16         private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
17         private readonly IConverter<TLink> _addressToUnaryNumberConverter;
18         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
19         private readonly IProperties<TLink, TLink, TLink> _propertyOperator;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

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

```

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

```

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

```

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

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences.Indexes
8  {
9      public class CachedFrequencyIncrementingSequenceIndex<TLink> : ISequenceIndex<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↳ EqualityComparer<TLink>.Default;
13
14         private readonly LinkFrequenciesCache<TLink> _cache;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
18             ↳ _cache = cache;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public bool Add(ICollection<TLink> sequence)
22         {
23             var indexed = true;
24             var i = sequence.Count;
25             while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
26                 ↳ { }
27             for (; i >= 1; i--)
28             {
29                 _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
30             }
31             return indexed;
32         }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         private bool IsIndexedWithIncrement(TLink source, TLink target)
36         {
37             var frequency = _cache.GetFrequency(source, target);
38             if (frequency == null)
39             {
40                 return false;
41             }
42             var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
43             if (indexed)
44             {
45                 _cache.IncrementFrequency(source, target);
46             }
47             return indexed;
48         }
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public bool MightContain(ICollection<TLink> sequence)
52         {
53             var indexed = true;
54             var i = sequence.Count;
55             while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
56             return indexed;
57         }
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         private bool IsIndexed(TLink source, TLink target)
61         {
62             var frequency = _cache.GetFrequency(source, target);
63             if (frequency == null)
64             {
65                 return false;
66             }
67             return !_equalityComparer.Equals(frequency.Frequency, default);
68         }
69     }
70 }

```

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

```

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

```

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```



```

33         links.GetOrCreate(sequence[i - 1], sequence[i]);
34     }
35     });
36 }
37     return indexed;
38 }
39
40 [MethodImpl(MethodImplOptions.AggressiveInlining)]
41 public bool MightContain(ICollection<TLink> sequence)
42 {
43     var links = _links.Unsync;
44     return _links.SyncRoot.ExecuteReadOperation(() =>
45     {
46         var indexed = true;
47         var i = sequence.Count;
48         while (--i >= 1 && (indexed =
49             ↳ !equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
50             ↳ sequence[i]), default))) { }
51         return indexed;
52     });
53 }

```

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

```

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

```

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

```

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

```

```

38         if (sequence.Length == 1)
39         {
40             return sequence;
41         }
42         return CreateAllVariants2Core(sequence, 0, (ulong)sequence.Length - 1);
43     });
44 }
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 private ulong[] CreateAllVariants2Core(ulong[] sequence, ulong startAt, ulong stopAt)
48 {
49     if ((stopAt - startAt) == 0)
50     {
51         return new[] { sequence[startAt] };
52     }
53     if ((stopAt - startAt) == 1)
54     {
55         return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt]) };
56     }
57     var variants = new ulong[Platform.Numbers.Math.Catalan(stopAt - startAt)];
58     var last = 0;
59     for (var splitter = startAt; splitter < stopAt; splitter++)
60     {
61         var left = CreateAllVariants2Core(sequence, startAt, splitter);
62         var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
63         for (var i = 0; i < left.Length; i++)
64         {
65             for (var j = 0; j < right.Length; j++)
66             {
67                 var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
68                 if (variant == Constants.Null)
69                 {
70                     throw new NotImplementedException("Creation cancellation is not
71                     ↳ implemented.");
72                 }
73                 variants[last++] = variant;
74             }
75         }
76     }
77     return variants;
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 public List<ulong> CreateAllVariants1(params ulong[] sequence)
82 {
83     return _sync.ExecuteWriteOperation(() =>
84     {
85         if (sequence.IsNullOrEmpty())
86         {
87             return new List<ulong>();
88         }
89         Links.Unsync.EnsureLinkExists(sequence);
90         if (sequence.Length == 1)
91         {
92             return new List<ulong> { sequence[0] };
93         }
94         var results = new
95             ↳ List<ulong>((int)Platform.Numbers.Math.Catalan((ulong)sequence.Length));
96         return CreateAllVariants1Core(sequence, results);
97     });
98 }
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
102 {
103     if (sequence.Length == 2)
104     {
105         var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
106         if (link == Constants.Null)
107         {
108             throw new NotImplementedException("Creation cancellation is not
109             ↳ implemented.");
110         }
111         results.Add(link);
112         return results;
113     }
114     var innerSequenceLength = sequence.Length - 1;
115     var innerSequence = new ulong[innerSequenceLength];

```

```

113     for (var li = 0; li < innerSequenceLength; li++)
114     {
115         var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
116         if (link == Constants.Null)
117         {
118             throw new NotImplementedException("Creation cancellation is not
119                 ↳ implemented.");
120         }
121         for (var isi = 0; isi < li; isi++)
122         {
123             innerSequence[isi] = sequence[isi];
124         }
125         innerSequence[li] = link;
126         for (var isi = li + 1; isi < innerSequenceLength; isi++)
127         {
128             innerSequence[isi] = sequence[isi + 1];
129         }
130         CreateAllVariants1Core(innerSequence, results);
131     }
132     return results;
133 }
134 #endregion
135
136 [MethodImpl(MethodImplOptions.AggressiveInlining)]
137 public HashSet<ulong> Each1(params ulong[] sequence)
138 {
139     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
140     Each1(link =>
141     {
142         if (!visitedLinks.Contains(link))
143         {
144             visitedLinks.Add(link); // изучить почему случаются повторы
145         }
146         return true;
147     }, sequence);
148     return visitedLinks;
149 }
150
151 [MethodImpl(MethodImplOptions.AggressiveInlining)]
152 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
153 {
154     if (sequence.Length == 2)
155     {
156         Links.Unsync.Each(sequence[0], sequence[1], handler);
157     }
158     else
159     {
160         var innerSequenceLength = sequence.Length - 1;
161         for (var li = 0; li < innerSequenceLength; li++)
162         {
163             var left = sequence[li];
164             var right = sequence[li + 1];
165             if (left == 0 && right == 0)
166             {
167                 continue;
168             }
169             var linkIndex = li;
170             ulong[] innerSequence = null;
171             Links.Unsync.Each(doublet =>
172             {
173                 if (innerSequence == null)
174                 {
175                     innerSequence = new ulong[innerSequenceLength];
176                     for (var isi = 0; isi < linkIndex; isi++)
177                     {
178                         innerSequence[isi] = sequence[isi];
179                     }
180                     for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
181                     {
182                         innerSequence[isi] = sequence[isi + 1];
183                     }
184                 }
185                 innerSequence[linkIndex] = doublet[Constants.IndexPart];
186                 Each1(handler, innerSequence);
187                 return Constants.Continue;
188             }, Constants.Any, left, right);
189         }
190     }
191 }

```

```

190     }
191 }
192
193 [MethodImpl(MethodImplOptions.AggressiveInlining)]
194 public HashSet<ulong> EachPart(params ulong[] sequence)
195 {
196     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
197     EachPartCore(link =>
198     {
199         var linkIndex = link[Constants.IndexPart];
200         if (!visitedLinks.Contains(linkIndex))
201         {
202             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
203         }
204         return Constants.Continue;
205     }, sequence);
206     return visitedLinks;
207 }
208
209 [MethodImpl(MethodImplOptions.AggressiveInlining)]
210 public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
211 {
212     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
213     EachPartCore(link =>
214     {
215         var linkIndex = link[Constants.IndexPart];
216         if (!visitedLinks.Contains(linkIndex))
217         {
218             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
219             return handler(new LinkAddress<LinkIndex>(linkIndex));
220         }
221         return Constants.Continue;
222     }, sequence);
223 }
224
225 [MethodImpl(MethodImplOptions.AggressiveInlining)]
226 private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
227     ↪ sequence)
228 {
229     if (sequence.IsNullOrEmpty())
230     {
231         return;
232     }
233     Links.EnsureLinkIsAnyOrExists(sequence);
234     if (sequence.Length == 1)
235     {
236         var link = sequence[0];
237         if (link > 0)
238         {
239             handler(new LinkAddress<LinkIndex>(link));
240         }
241         else
242         {
243             Links.Each(Constants.Any, Constants.Any, handler);
244         }
245     }
246     else if (sequence.Length == 2)
247     {
248         //_links.Each(sequence[0], sequence[1], handler);
249         // o_|      x_o ...
250         // x_|      |__|
251         Links.Each(sequence[1], Constants.Any, doublet =>
252         {
253             var match = Links.SearchOrDefault(sequence[0], doublet);
254             if (match != Constants.Null)
255             {
256                 handler(new LinkAddress<LinkIndex>(match));
257             }
258             return true;
259         });
260         // |_x      ... x_o
261         // |_o      |__|
262         Links.Each(Constants.Any, sequence[0], doublet =>
263         {
264             var match = Links.SearchOrDefault(doublet, sequence[1]);
265             if (match != 0)
266             {
267                 handler(new LinkAddress<LinkIndex>(match));
268             }
269         });
270     }
271 }

```

```

267         }
268         return true;
269     });
270     //      ..x o..
271     //      |__|
272     PartialStepRight(x => handler(x), sequence[0], sequence[1]);
273 }
274 else
275 {
276     throw new NotImplementedException();
277 }
278 }
279
280 [MethodImpl(MethodImplOptions.AggressiveInlining)]
281 private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
282 {
283     Links.Unsync.Each(Constants.Any, left, doublet =>
284     {
285         StepRight(handler, doublet, right);
286         if (left != doublet)
287         {
288             PartialStepRight(handler, doublet, right);
289         }
290         return true;
291     });
292 }
293
294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
296 {
297     Links.Unsync.Each(left, Constants.Any, rightStep =>
298     {
299         TryStepRightUp(handler, right, rightStep);
300         return true;
301     });
302 }
303
304 [MethodImpl(MethodImplOptions.AggressiveInlining)]
305 private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
306 ↪ stepFrom)
307 {
308     var upStep = stepFrom;
309     var firstSource = Links.Unsync.GetTarget(upStep);
310     while (firstSource != right && firstSource != upStep)
311     {
312         upStep = firstSource;
313         firstSource = Links.Unsync.GetSource(upStep);
314     }
315     if (firstSource == right)
316     {
317         handler(new LinkAddress<LinkIndex>(stepFrom));
318     }
319 }
320 // TODO: Test
321 [MethodImpl(MethodImplOptions.AggressiveInlining)]
322 private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
323 {
324     Links.Unsync.Each(right, Constants.Any, doublet =>
325     {
326         StepLeft(handler, left, doublet);
327         if (right != doublet)
328         {
329             PartialStepLeft(handler, left, doublet);
330         }
331         return true;
332     });
333 }
334
335 [MethodImpl(MethodImplOptions.AggressiveInlining)]
336 private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
337 {
338     Links.Unsync.Each(Constants.Any, right, leftStep =>
339     {
340         TryStepLeftUp(handler, left, leftStep);
341         return true;
342     });
343 }
344

```

```

345 [MethodImpl(MethodImplOptions.AggressiveInlining)]
346 private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
347 {
348     var upStep = stepFrom;
349     var firstTarget = Links.Unsync.GetSource(upStep);
350     while (firstTarget != left && firstTarget != upStep)
351     {
352         upStep = firstTarget;
353         firstTarget = Links.Unsync.GetTarget(upStep);
354     }
355     if (firstTarget == left)
356     {
357         handler(new LinkAddress<LinkIndex>(stepFrom));
358     }
359 }
360
361 [MethodImpl(MethodImplOptions.AggressiveInlining)]
362 private bool StartsWith(ulong sequence, ulong link)
363 {
364     var upStep = sequence;
365     var firstSource = Links.Unsync.GetSource(upStep);
366     while (firstSource != link && firstSource != upStep)
367     {
368         upStep = firstSource;
369         firstSource = Links.Unsync.GetSource(upStep);
370     }
371     return firstSource == link;
372 }
373
374 [MethodImpl(MethodImplOptions.AggressiveInlining)]
375 private bool EndsWith(ulong sequence, ulong link)
376 {
377     var upStep = sequence;
378     var lastTarget = Links.Unsync.GetTarget(upStep);
379     while (lastTarget != link && lastTarget != upStep)
380     {
381         upStep = lastTarget;
382         lastTarget = Links.Unsync.GetTarget(upStep);
383     }
384     return lastTarget == link;
385 }
386
387 [MethodImpl(MethodImplOptions.AggressiveInlining)]
388 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
389 {
390     return _sync.ExecuteReadOperation(() =>
391     {
392         var results = new List<ulong>();
393         if (sequence.Length > 0)
394         {
395             Links.EnsureLinkExists(sequence);
396             var firstElement = sequence[0];
397             if (sequence.Length == 1)
398             {
399                 results.Add(firstElement);
400                 return results;
401             }
402             if (sequence.Length == 2)
403             {
404                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
405                 if (doublet != Constants.Null)
406                 {
407                     results.Add(doublet);
408                 }
409                 return results;
410             }
411             var linksInSequence = new HashSet<ulong>(sequence);
412             void handler(IList<LinkIndex> result)
413             {
414                 var resultIndex = result[Links.Constants.IndexPart];
415                 var filterPosition = 0;
416                 StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
417                     ↪ Links.Unsync.GetTarget,
418                     x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
419                     ↪ x =>
420                     {
421                         if (filterPosition == sequence.Length)
422                         {
423                             filterPosition = -2; // Длиннее чем нужно

```

```

422         return false;
423     }
424     if (x != sequence[filterPosition])
425     {
426         filterPosition = -1;
427         return false; // Начинается иначе
428     }
429     filterPosition++;
430
431     return true;
432     });
433     if (filterPosition == sequence.Length)
434     {
435         results.Add(resultIndex);
436     }
437 }
438 if (sequence.Length >= 2)
439 {
440     StepRight(handler, sequence[0], sequence[1]);
441 }
442 var last = sequence.Length - 2;
443 for (var i = 1; i < last; i++)
444 {
445     PartialStepRight(handler, sequence[i], sequence[i + 1]);
446 }
447 if (sequence.Length >= 3)
448 {
449     StepLeft(handler, sequence[sequence.Length - 2],
450         ↪ sequence[sequence.Length - 1]);
451 }
452 return results;
453 });
454 }
455
456 [MethodImpl(MethodImplOptions.AggressiveInlining)]
457 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
458 {
459     return _sync.ExecuteReadOperation(() =>
460     {
461         var results = new HashSet<ulong>();
462         if (sequence.Length > 0)
463         {
464             Links.EnsureLinkExists(sequence);
465             var firstElement = sequence[0];
466             if (sequence.Length == 1)
467             {
468                 results.Add(firstElement);
469                 return results;
470             }
471             if (sequence.Length == 2)
472             {
473                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
474                 if (doublet != Constants.Null)
475                 {
476                     results.Add(doublet);
477                 }
478                 return results;
479             }
480             var matcher = new Matcher(this, sequence, results, null);
481             if (sequence.Length >= 2)
482             {
483                 StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
484             }
485             var last = sequence.Length - 2;
486             for (var i = 1; i < last; i++)
487             {
488                 PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
489                     ↪ sequence[i + 1]);
490             }
491             if (sequence.Length >= 3)
492             {
493                 StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
494                     ↪ sequence[sequence.Length - 1]);
495             }
496             return results;
497         }
498     });

```

```

497 }
498
499 public const int MaxSequenceFormatSize = 200;
500
501 [MethodImpl(MethodImplOptions.AggressiveInlining)]
502 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
503     => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
504
505 [MethodImpl(MethodImplOptions.AggressiveInlining)]
506 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
507     elementToString, bool insertComma, params LinkIndex[] knownElements) =>
508     Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
509         elementToString, insertComma, knownElements));
510
511 [MethodImpl(MethodImplOptions.AggressiveInlining)]
512 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
513     Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
514     LinkIndex[] knownElements)
515 {
516     var linksInSequence = new HashSet<ulong>(knownElements);
517     //var entered = new HashSet<ulong>();
518     var sb = new StringBuilder();
519     sb.Append('{');
520     if (links.Exists(sequenceLink))
521     {
522         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
523             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
524             entered.AddAndReturnVoid, x => { }, entered.DoNotContains
525             {
526                 if (insertComma && sb.Length > 1)
527                 {
528                     sb.Append(',');
529                 }
530                 //if (entered.Contains(element))
531                 //{
532                     sb.Append('{');
533                     elementToString(sb, element);
534                     sb.Append('}');
535                 //}
536                 //else
537                 elementToString(sb, element);
538                 if (sb.Length < MaxSequenceFormatSize)
539                 {
540                     return true;
541                 }
542                 sb.Append(insertComma ? ", ..." : "...");
543                 return false;
544             });
545     }
546     sb.Append('}');
547     return sb.ToString();
548 }
549
550 [MethodImpl(MethodImplOptions.AggressiveInlining)]
551 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
552     knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
553     knownElements);
554
555 [MethodImpl(MethodImplOptions.AggressiveInlining)]
556 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
557     LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
558     Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
559         sequenceLink, elementToString, insertComma, knownElements));
560
561 [MethodImpl(MethodImplOptions.AggressiveInlining)]
562 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
563     Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
564     LinkIndex[] knownElements)
565 {
566     var linksInSequence = new HashSet<ulong>(knownElements);
567     var entered = new HashSet<ulong>();
568     var sb = new StringBuilder();
569     sb.Append('{');
570     if (links.Exists(sequenceLink))
571     {
572         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
573             x => linksInSequence.Contains(x) || links.IsFullPoint(x),
574             entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>

```



```

560     {
561         if (insertComma && sb.Length > 1)
562         {
563             sb.Append(',');
564         }
565         if (entered.Contains(element))
566         {
567             sb.Append('{');
568             elementToString(sb, element);
569             sb.Append('}');
570         }
571         else
572         {
573             elementToString(sb, element);
574         }
575         if (sb.Length < MaxSequenceFormatSize)
576         {
577             return true;
578         }
579         sb.Append(insertComma ? ", ..." : "...");
580         return false;
581     });
582 }
583 sb.Append('}');
584 return sb.ToString();
585 }
586
587 [MethodImpl(MethodImplOptions.AggressiveInlining)]
588 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
589 {
590     return _sync.ExecuteReadOperation(() =>
591     {
592         if (sequence.Length > 0)
593         {
594             Links.EnsureLinkExists(sequence);
595             var results = new HashSet<ulong>();
596             for (var i = 0; i < sequence.Length; i++)
597             {
598                 AllUsagesCore(sequence[i], results);
599             }
600             var filteredResults = new List<ulong>();
601             var linksInSequence = new HashSet<ulong>(sequence);
602             foreach (var result in results)
603             {
604                 var filterPosition = -1;
605                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
606                     ↪ Links.Unsync.GetTarget,
607                     ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
608                     ↪ x =>
609                     {
610                         if (filterPosition == (sequence.Length - 1))
611                         {
612                             return false;
613                         }
614                         if (filterPosition >= 0)
615                         {
616                             if (x == sequence[filterPosition + 1])
617                             {
618                                 filterPosition++;
619                             }
620                             else
621                             {
622                                 return false;
623                             }
624                         }
625                         if (filterPosition < 0)
626                         {
627                             if (x == sequence[0])
628                             {
629                                 filterPosition = 0;
630                             }
631                         }
632                         return true;
633                     });
634             if (filterPosition == (sequence.Length - 1))
635             {
636                 filteredResults.Add(result);
637             }
638         }
639     });
640 }

```

```

637         return filteredResults;
638     }
639     return new List<ulong>();
640 });
641 }
642
643 [MethodImpl(MethodImplOptions.AggressiveInlining)]
644 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
645 {
646     return _sync.ExecuteReadOperation(() =>
647     {
648         if (sequence.Length > 0)
649         {
650             Links.EnsureLinkExists(sequence);
651             var results = new HashSet<ulong>();
652             for (var i = 0; i < sequence.Length; i++)
653             {
654                 AllUsagesCore(sequence[i], results);
655             }
656             var filteredResults = new HashSet<ulong>();
657             var matcher = new Matcher(this, sequence, filteredResults, null);
658             matcher.AddAllPartialMatchedToResults(results);
659             return filteredResults;
660         }
661         return new HashSet<ulong>();
662     });
663 }
664
665 [MethodImpl(MethodImplOptions.AggressiveInlining)]
666 public bool GetAllPartiallyMatchingSequences2(Func<ILinkIndex>, LinkIndex> handler,
667     ↪ params ulong[] sequence)
668 {
669     return _sync.ExecuteReadOperation(() =>
670     {
671         if (sequence.Length > 0)
672         {
673             Links.EnsureLinkExists(sequence);
674
675             var results = new HashSet<ulong>();
676             var filteredResults = new HashSet<ulong>();
677             var matcher = new Matcher(this, sequence, filteredResults, handler);
678             for (var i = 0; i < sequence.Length; i++)
679             {
680                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
681                 {
682                     return false;
683                 }
684             }
685             return true;
686         }
687         return true;
688     });
689 }
690
691 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
692 //{{
693 //    return Sync.ExecuteReadOperation(() =>
694 //    {
695 //        if (sequence.Length > 0)
696 //        {
697 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
698 //
699 //            var firstResults = new HashSet<ulong>();
700 //            var lastResults = new HashSet<ulong>();
701 //
702 //            var first = sequence.First(x => x != LinksConstants.Any);
703 //            var last = sequence.Last(x => x != LinksConstants.Any);
704 //
705 //            AllUsagesCore(first, firstResults);
706 //            AllUsagesCore(last, lastResults);
707 //
708 //            firstResults.IntersectWith(lastResults);
709 //
710 //            //for (var i = 0; i < sequence.Length; i++)
711 //            //    AllUsagesCore(sequence[i], results);
712 //
713 //            var filteredResults = new HashSet<ulong>();
714 //            var matcher = new Matcher(this, sequence, filteredResults, null);
715 //            matcher.AddAllPartialMatchedToResults(firstResults);

```

```

715 //         return filteredResults;
716 //     }
717
718 //     return new HashSet<ulong>();
719 // });
720 //}
721
722 [MethodImpl(MethodImplOptions.AggressiveInlining)]
723 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
724 {
725     return _sync.ExecuteReadOperation(() =>
726     {
727         if (sequence.Length > 0)
728         {
729             ILinksExtensions.EnsureLinkIsAnyOrExists(Links, sequence);
730             var firstResults = new HashSet<ulong>();
731             var lastResults = new HashSet<ulong>();
732             var first = sequence.First(x => x != Constants.Any);
733             var last = sequence.Last(x => x != Constants.Any);
734             AllUsagesCore(first, firstResults);
735             AllUsagesCore(last, lastResults);
736             firstResults.IntersectWith(lastResults);
737             //for (var i = 0; i < sequence.Length; i++)
738             //    AllUsagesCore(sequence[i], results);
739             var filteredResults = new HashSet<ulong>();
740             var matcher = new Matcher(this, sequence, filteredResults, null);
741             matcher.AddAllPartialMatchedToResults(firstResults);
742             return filteredResults;
743         }
744         return new HashSet<ulong>();
745     });
746 }
747
748 [MethodImpl(MethodImplOptions.AggressiveInlining)]
749 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
750     ↪ IList<ulong> sequence)
751 {
752     return _sync.ExecuteReadOperation(() =>
753     {
754         if (sequence.Count > 0)
755         {
756             Links.EnsureLinkExists(sequence);
757             var results = new HashSet<LinkIndex>();
758             //var nextResults = new HashSet<ulong>();
759             //for (var i = 0; i < sequence.Length; i++)
760             //{
761             //    AllUsagesCore(sequence[i], nextResults);
762             //    if (results.IsNullOrEmpty())
763             //    {
764             //        results = nextResults;
765             //        nextResults = new HashSet<ulong>();
766             //    }
767             //    else
768             //    {
769             //        results.IntersectWith(nextResults);
770             //        nextResults.Clear();
771             //    }
772             //}
773             var collector1 = new AllUsagesCollector1(Links.Unsync, results);
774             collector1.Collect(Links.Unsync.GetLink(sequence[0]));
775             var next = new HashSet<ulong>();
776             for (var i = 1; i < sequence.Count; i++)
777             {
778                 var collector = new AllUsagesCollector1(Links.Unsync, next);
779                 collector.Collect(Links.Unsync.GetLink(sequence[i]));
780
781                 results.IntersectWith(next);
782                 next.Clear();
783             }
784             var filteredResults = new HashSet<ulong>();
785             var matcher = new Matcher(this, sequence, filteredResults, null,
786                 ↪ readAsElements);
787             matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
788                 ↪ x)); // OrderBy is a Hack
789             return filteredResults;
790         }
791         return new HashSet<ulong>();
792     });
793 }

```

```
// Does not work
//public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
//    params ulong[] sequence)
//{
//    var visited = new HashSet<ulong>();
//    var results = new HashSet<ulong>();
//    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
//    true; }, readAsElements);
//    var last = sequence.Length - 1;
//    for (var i = 0; i < last; i++)
//    {
//        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
//    }
//    return results;
//}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
{
    return _sync.ExecuteReadOperation(() =>
    {
        if (sequence.Length > 0)
        {
            Links.EnsureLinkExists(sequence);
            //var firstElement = sequence[0];
            //if (sequence.Length == 1)
            //{
            //    //results.Add(firstElement);
            //    return results;
            //}
            //if (sequence.Length == 2)
            //{
            //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
            //    //if (doublet != Doublets.Links.Null)
            //    //    results.Add(doublet);
            //    return results;
            //}
            //var lastElement = sequence[sequence.Length - 1];
            //Func<ulong, bool> handler = x =>
            //{
            //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
            //        results.Add(x);
            //    return true;
            //};
            //if (sequence.Length >= 2)
            //    StepRight(handler, sequence[0], sequence[1]);
            //var last = sequence.Length - 2;
            //for (var i = 1; i < last; i++)
            //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
            //if (sequence.Length >= 3)
            //    StepLeft(handler, sequence[sequence.Length - 2],
            //        sequence[sequence.Length - 1]);
            //if (sequence.Length == 1)
            //    throw new NotImplementedException(); // all sequences, containing
            //    this element?
            //if (sequence.Length == 2)
            //{
            //    var results = new List<ulong>();
            //    PartialStepRight(results.Add, sequence[0], sequence[1]);
            //    return results;
            //}
            //var matches = new List<List<ulong>>();
            //var last = sequence.Length - 1;
            //for (var i = 0; i < last; i++)
            //{
            //    var results = new List<ulong>();
            //    //StepRight(results.Add, sequence[i], sequence[i + 1]);
            //    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
            //    if (results.Count > 0)
            //        matches.Add(results);
            //    else
            //        return results;
            //    if (matches.Count == 2)
```

```

862         {
863             var merged = new List<ulong>();
864             for (var j = 0; j < matches[0].Count; j++)
865                 for (var k = 0; k < matches[1].Count; k++)
866                     CloseInnerConnections(merged.Add, matches[0][j],
867                                     ↪ matches[1][k]);
868             if (merged.Count > 0)
869                 matches = new List<List<ulong>> { merged };
870             else
871                 return new List<ulong>();
872         }
873     }
874     if (matches.Count > 0)
875     {
876         var usages = new HashSet<ulong>();
877         for (int i = 0; i < sequence.Length; i++)
878         {
879             AllUsagesCore(sequence[i], usages);
880         }
881         //for (int i = 0; i < matches[0].Count; i++)
882         //    AllUsagesCore(matches[0][i], usages);
883         //usages.UnionWith(matches[0]);
884         return usages.ToList();
885     }
886     var firstLinkUsages = new HashSet<ulong>();
887     AllUsagesCore(sequence[0], firstLinkUsages);
888     firstLinkUsages.Add(sequence[0]);
889     //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
890     ↪ sequence[0] }; // or all sequences, containing this element?
891     //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
892     ↪ 1).ToList();
893     var results = new HashSet<ulong>();
894     foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
895     ↪ firstLinkUsages, 1))
896     {
897         AllUsagesCore(match, results);
898     }
899     return results.ToList();
900 }
901 }
902 return new List<ulong>();
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 }

```

/// <remarks>
 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
 /// </remarks>
 [MethodImpl(MethodImplOptions.AggressiveInlining)]
 public HashSet<ulong> AllUsages(ulong link)
 {
 return _sync.ExecuteReadOperation(() =>
 {
 var usages = new HashSet<ulong>();
 AllUsagesCore(link, usages);
 return usages;
 });
 }
}

// При сборе всех использований (последовательностей) можно сохранять обратный путь к
 ↪ той связи с которой начинался поиск (STTTSSSTT),
 // причём достаточно одного бита для хранения перехода влево или вправо
 [MethodImpl(MethodImplOptions.AggressiveInlining)]
 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
 {
 bool handler(ulong doublet)
 {
 if (usages.Add(doublet))
 {
 AllUsagesCore(doublet, usages);
 }
 return true;
 }
 Links.Unsync.Each(link, Constants.Any, handler);
 Links.Unsync.Each(Constants.Any, link, handler);
 }
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
 public HashSet<ulong> AllBottomUsages(ulong link)

```

934 {
935     return _sync.ExecuteReadOperation(() =>
936     {
937         var visits = new HashSet<ulong>();
938         var usages = new HashSet<ulong>();
939         AllBottomUsagesCore(link, visits, usages);
940         return usages;
941     });
942 }
943
944 [MethodImpl(MethodImplOptions.AggressiveInlining)]
945 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
↪ usages)
946 {
947     bool handler(ulong doublet)
948     {
949         if (visits.Add(doublet))
950         {
951             AllBottomUsagesCore(doublet, visits, usages);
952         }
953         return true;
954     }
955     if (Links.Unsync.Count(Constants.Any, link) == 0)
956     {
957         usages.Add(link);
958     }
959     else
960     {
961         Links.Unsync.Each(link, Constants.Any, handler);
962         Links.Unsync.Each(Constants.Any, link, handler);
963     }
964 }
965
966 [MethodImpl(MethodImplOptions.AggressiveInlining)]
967 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
968 {
969     if (Options.UseSequenceMarker)
970     {
971         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
↪ Options.MarkedSequenceMatcher, symbol);
972         return counter.Count();
973     }
974     else
975     {
976         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
↪ symbol);
977         return counter.Count();
978     }
979 }
980
981 [MethodImpl(MethodImplOptions.AggressiveInlining)]
982 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
↪ LinkIndex> outerHandler)
983 {
984     bool handler(ulong doublet)
985     {
986         if (usages.Add(doublet))
987         {
988             if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
989             {
990                 return false;
991             }
992             if (!AllUsagesCore1(doublet, usages, outerHandler))
993             {
994                 return false;
995             }
996         }
997         return true;
998     }
999     return Links.Unsync.Each(link, Constants.Any, handler)
1000         && Links.Unsync.Each(Constants.Any, link, handler);
1001 }
1002
1003 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1004 public void CalculateAllUsages(ulong[] totals)
1005 {
1006     var calculator = new AllUsagesCalculator(Links, totals);
1007     calculator.Calculate();

```

```

1008     }
1009
1010     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1011     public void CalculateAllUsages2(ulong[] totals)
1012     {
1013         var calculator = new AllUsagesCalculator2(Links, totals);
1014         calculator.Calculate();
1015     }
1016
1017     private class AllUsagesCalculator
1018     {
1019         private readonly SynchronizedLinks<ulong> _links;
1020         private readonly ulong[] _totals;
1021
1022         [MethodImpl(MethodImplOptions.AggressiveInlining)]
1023         public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
1024         {
1025             _links = links;
1026             _totals = totals;
1027         }
1028
1029         [MethodImpl(MethodImplOptions.AggressiveInlining)]
1030         public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1031             ↪ CalculateCore);
1032
1033         [MethodImpl(MethodImplOptions.AggressiveInlining)]
1034         private bool CalculateCore(ulong link)
1035         {
1036             if (_totals[link] == 0)
1037             {
1038                 var total = 1UL;
1039                 _totals[link] = total;
1040                 var visitedChildren = new HashSet<ulong>();
1041                 bool linkCalculator(ulong child)
1042                 {
1043                     if (link != child && visitedChildren.Add(child))
1044                     {
1045                         total += _totals[child] == 0 ? 1 : _totals[child];
1046                     }
1047                     return true;
1048                 }
1049                 _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1050                 _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1051                 _totals[link] = total;
1052             }
1053             return true;
1054         }
1055     }
1056
1057     private class AllUsagesCalculator2
1058     {
1059         private readonly SynchronizedLinks<ulong> _links;
1060         private readonly ulong[] _totals;
1061
1062         [MethodImpl(MethodImplOptions.AggressiveInlining)]
1063         public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1064         {
1065             _links = links;
1066             _totals = totals;
1067         }
1068
1069         [MethodImpl(MethodImplOptions.AggressiveInlining)]
1070         public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1071             ↪ CalculateCore);
1072
1073         [MethodImpl(MethodImplOptions.AggressiveInlining)]
1074         private bool IsElement(ulong link)
1075         {
1076             // _linksInSequence.Contains(link) ||
1077             return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1078                 ↪ link;
1079         }
1080
1081         [MethodImpl(MethodImplOptions.AggressiveInlining)]
1082         private bool CalculateCore(ulong link)
1083         {
1084             // TODO: Проработать защиту от заикливания
1085             // Основано на SequenceWalker.WalkLeft
1086             Func<ulong, ulong> getSource = _links.Unsync.GetSource;

```

```

1084 Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1085 Func<ulong, bool> isElement = IsElement;
1086 void visitLeaf(ulong parent)
1087 {
1088     if (link != parent)
1089     {
1090         _totals[parent]++;
1091     }
1092 }
1093 void visitNode(ulong parent)
1094 {
1095     if (link != parent)
1096     {
1097         _totals[parent]++;
1098     }
1099 }
1100 var stack = new Stack();
1101 var element = link;
1102 if (isElement(element))
1103 {
1104     visitLeaf(element);
1105 }
1106 else
1107 {
1108     while (true)
1109     {
1110         if (isElement(element))
1111         {
1112             if (stack.Count == 0)
1113             {
1114                 break;
1115             }
1116             element = stack.Pop();
1117             var source = getSource(element);
1118             var target = getTarget(element);
1119             // Обработка элемента
1120             if (isElement(target))
1121             {
1122                 visitLeaf(target);
1123             }
1124             if (isElement(source))
1125             {
1126                 visitLeaf(source);
1127             }
1128             element = source;
1129         }
1130         else
1131         {
1132             stack.Push(element);
1133             visitNode(element);
1134             element = getTarget(element);
1135         }
1136     }
1137 }
1138 _totals[link]++;
1139 return true;
1140 }
1141 }
1142
1143 private class AllUsagesCollector
1144 {
1145     private readonly ILinks<ulong> _links;
1146     private readonly HashSet<ulong> _usages;
1147
1148     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1149     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1150     {
1151         _links = links;
1152         _usages = usages;
1153     }
1154
1155     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1156     public bool Collect(ulong link)
1157     {
1158         if (_usages.Add(link))
1159         {
1160             _links.Each(link, _links.Constants.Any, Collect);
1161             _links.Each(_links.Constants.Any, link, Collect);
1162         }
1163     }

```



```

1163         return true;
1164     }
1165 }
1166
1167 private class AllUsagesCollector1
1168 {
1169     private readonly ILinks<ulong> _links;
1170     private readonly HashSet<ulong> _usages;
1171     private readonly ulong _continue;
1172
1173     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1174     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1175     {
1176         _links = links;
1177         _usages = usages;
1178         _continue = _links.Constants.Continue;
1179     }
1180
1181     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1182     public ulong Collect(IList<ulong> link)
1183     {
1184         var linkIndex = _links.GetIndex(link);
1185         if (_usages.Add(linkIndex))
1186         {
1187             _links.Each(Collect, _links.Constants.Any, linkIndex);
1188         }
1189         return _continue;
1190     }
1191 }
1192
1193 private class AllUsagesCollector2
1194 {
1195     private readonly ILinks<ulong> _links;
1196     private readonly BitString _usages;
1197
1198     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1199     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1200     {
1201         _links = links;
1202         _usages = usages;
1203     }
1204
1205     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1206     public bool Collect(ulong link)
1207     {
1208         if (_usages.Add((long)link))
1209         {
1210             _links.Each(link, _links.Constants.Any, Collect);
1211             _links.Each(_links.Constants.Any, link, Collect);
1212         }
1213         return true;
1214     }
1215 }
1216
1217 private class AllUsagesIntersectingCollector
1218 {
1219     private readonly SynchronizedLinks<ulong> _links;
1220     private readonly HashSet<ulong> _intersectWith;
1221     private readonly HashSet<ulong> _usages;
1222     private readonly HashSet<ulong> _enter;
1223
1224     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1225     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
↵ intersectWith, HashSet<ulong> usages)
1226     {
1227         _links = links;
1228         _intersectWith = intersectWith;
1229         _usages = usages;
1230         _enter = new HashSet<ulong>(); // защита от зацикливания
1231     }
1232
1233     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1234     public bool Collect(ulong link)
1235     {
1236         if (_enter.Add(link))
1237         {
1238             if (_intersectWith.Contains(link))
1239             {
1240                 _usages.Add(link);
1241             }

```

```

1242         _links.Unsync.Each(link, _links.Constants.Any, Collect);
1243         _links.Unsync.Each(_links.Constants.Any, link, Collect);
1244     }
1245     return true;
1246 }
1247 }
1248
1249 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1250 private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
    ↪ right)
1251 {
1252     TryStepLeftUp(handler, left, right);
1253     TryStepRightUp(handler, right, left);
1254 }
1255
1256 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1257 private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
    ↪ right)
1258 {
1259     // Direct
1260     if (left == right)
1261     {
1262         handler(new LinkAddress<LinkIndex>(left));
1263     }
1264     var doublet = Links.Unsync.SearchOrDefault(left, right);
1265     if (doublet != Constants.Null)
1266     {
1267         handler(new LinkAddress<LinkIndex>(doublet));
1268     }
1269     // Inner
1270     CloseInnerConnections(handler, left, right);
1271     // Outer
1272     StepLeft(handler, left, right);
1273     StepRight(handler, left, right);
1274     PartialStepRight(handler, left, right);
1275     PartialStepLeft(handler, left, right);
1276 }
1277
1278 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1279 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
    ↪ HashSet<ulong> previousMatchings, long startAt)
1280 {
1281     if (startAt >= sequence.Length) // ?
1282     {
1283         return previousMatchings;
1284     }
1285     var secondLinkUsages = new HashSet<ulong>();
1286     AllUsagesCore(sequence[startAt], secondLinkUsages);
1287     secondLinkUsages.Add(sequence[startAt]);
1288     var matchings = new HashSet<ulong>();
1289     var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
1290     //for (var i = 0; i < previousMatchings.Count; i++)
1291     foreach (var secondLinkUsage in secondLinkUsages)
1292     {
1293         foreach (var previousMatching in previousMatchings)
1294         {
1295             //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
1296             ↪ secondLinkUsage);
1297             StepRight(filler.AddFirstAndReturnConstant, previousMatching,
1298             ↪ secondLinkUsage);
1299             TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
1300             ↪ previousMatching);
1301             //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1302             ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1303             ↪ желаемым результатам.
1304             PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
1305             ↪ secondLinkUsage);
1306         }
1307     }
1308     if (matchings.Count == 0)
1309     {
1310         return matchings;
1311     }
1312     return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1313 }
1314
1315 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

1310 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
    ↳ links, params ulong[] sequence)
1311 {
1312     if (sequence == null)
1313     {
1314         return;
1315     }
1316     for (var i = 0; i < sequence.Length; i++)
1317     {
1318         if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
            ↳ !links.Exists(sequence[i]))
1319         {
1320             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                ↳ $"patternSequence[{i}]");
1321         }
1322     }
1323 }
1324
1325 // Pattern Matching -> Key To Triggers
1326 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1327 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1328 {
1329     return _sync.ExecuteReadOperation(() =>
1330     {
1331         patternSequence = Simplify(patternSequence);
1332         if (patternSequence.Length > 0)
1333         {
1334             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1335             var uniqueSequenceElements = new HashSet<ulong>();
1336             for (var i = 0; i < patternSequence.Length; i++)
1337             {
1338                 if (patternSequence[i] != Constants.Any && patternSequence[i] !=
                    ↳ ZeroOrMany)
1339                 {
1340                     uniqueSequenceElements.Add(patternSequence[i]);
1341                 }
1342             }
1343             var results = new HashSet<ulong>();
1344             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1345             {
1346                 AllUsagesCore(uniqueSequenceElement, results);
1347             }
1348             var filteredResults = new HashSet<ulong>();
1349             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1350             matcher.AddAllPatternMatchedToResults(results);
1351             return filteredResults;
1352         }
1353         return new HashSet<ulong>();
1354     });
1355 }
1356
1357 // Найти все возможные связи между указанным списком связей.
1358 // Находит связи между всеми указанными связями в любом порядке.
1359 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
    ↳ несколько раз в последовательности)
1360 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1361 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1362 {
1363     return _sync.ExecuteReadOperation(() =>
1364     {
1365         var results = new HashSet<ulong>();
1366         if (linksToConnect.Length > 0)
1367         {
1368             Links.EnsureLinkExists(linksToConnect);
1369             AllUsagesCore(linksToConnect[0], results);
1370             for (var i = 1; i < linksToConnect.Length; i++)
1371             {
1372                 var next = new HashSet<ulong>();
1373                 AllUsagesCore(linksToConnect[i], next);
1374                 results.IntersectWith(next);
1375             }
1376         }
1377         return results;
1378     });
1379 }
1380
1381 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1382 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)

```

```

1383 {
1384     return _sync.ExecuteReadOperation(() =>
1385     {
1386         var results = new HashSet<ulong>();
1387         if (linksToConnect.Length > 0)
1388         {
1389             Links.EnsureLinkExists(linksToConnect);
1390             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1391             collector1.Collect(linksToConnect[0]);
1392             var next = new HashSet<ulong>();
1393             for (var i = 1; i < linksToConnect.Length; i++)
1394             {
1395                 var collector = new AllUsagesCollector(Links.Unsync, next);
1396                 collector.Collect(linksToConnect[i]);
1397                 results.IntersectWith(next);
1398                 next.Clear();
1399             }
1400         }
1401         return results;
1402     });
1403 }
1404
1405 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1406 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1407 {
1408     return _sync.ExecuteReadOperation(() =>
1409     {
1410         var results = new HashSet<ulong>();
1411         if (linksToConnect.Length > 0)
1412         {
1413             Links.EnsureLinkExists(linksToConnect);
1414             var collector1 = new AllUsagesCollector(Links, results);
1415             collector1.Collect(linksToConnect[0]);
1416             //AllUsagesCore(linksToConnect[0], results);
1417             for (var i = 1; i < linksToConnect.Length; i++)
1418             {
1419                 var next = new HashSet<ulong>();
1420                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1421                 collector.Collect(linksToConnect[i]);
1422                 //AllUsagesCore(linksToConnect[i], next);
1423                 //results.IntersectWith(next);
1424                 results = next;
1425             }
1426         }
1427         return results;
1428     });
1429 }
1430
1431 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1432 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1433 {
1434     return _sync.ExecuteReadOperation(() =>
1435     {
1436         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1437         ⇨ BitArray((int)_links.Total + 1);
1438         if (linksToConnect.Length > 0)
1439         {
1440             Links.EnsureLinkExists(linksToConnect);
1441             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1442             collector1.Collect(linksToConnect[0]);
1443             for (var i = 1; i < linksToConnect.Length; i++)
1444             {
1445                 var next = new BitString((long)Links.Unsync.Count() + 1); //new
1446                 ⇨ BitArray((int)_links.Total + 1);
1447                 var collector = new AllUsagesCollector2(Links.Unsync, next);
1448                 collector.Collect(linksToConnect[i]);
1449                 results = results.And(next);
1450             }
1451         }
1452         return results.GetSetUInt64Indices();
1453     });
1454 }
1455
1456 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1457 private static ulong[] Simplify(ulong[] sequence)
1458 {
1459     // Считаем новый размер последовательности
1460     long newLength = 0;

```

```

1459 var zeroOrManyStepped = false;
1460 for (var i = 0; i < sequence.Length; i++)
1461 {
1462     if (sequence[i] == ZeroOrMany)
1463     {
1464         if (zeroOrManyStepped)
1465         {
1466             continue;
1467         }
1468         zeroOrManyStepped = true;
1469     }
1470     else
1471     {
1472         //if (zeroOrManyStepped) Is it efficient?
1473         zeroOrManyStepped = false;
1474     }
1475     newLength++;
1476 }
1477 // Строим новую последовательность
1478 zeroOrManyStepped = false;
1479 var newSequence = new ulong[newLength];
1480 long j = 0;
1481 for (var i = 0; i < sequence.Length; i++)
1482 {
1483     //var current = zeroOrManyStepped;
1484     //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1485     //if (current && zeroOrManyStepped)
1486     //    continue;
1487     //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1488     //if (zeroOrManyStepped && newZeroOrManyStepped)
1489     //    continue;
1490     //zeroOrManyStepped = newZeroOrManyStepped;
1491     if (sequence[i] == ZeroOrMany)
1492     {
1493         if (zeroOrManyStepped)
1494         {
1495             continue;
1496         }
1497         zeroOrManyStepped = true;
1498     }
1499     else
1500     {
1501         //if (zeroOrManyStepped) Is it efficient?
1502         zeroOrManyStepped = false;
1503     }
1504     newSequence[j++] = sequence[i];
1505 }
1506 return newSequence;
1507 }
1508
1509 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1510 public static void TestSimplify()
1511 {
1512     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
1513     ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1514     var simplifiedSequence = Simplify(sequence);
1515 }
1516
1517 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1518 public List<ulong> GetSimilarSequences() => new List<ulong>();
1519
1520 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1521 public void Prediction()
1522 {
1523     // _links
1524     // sequences
1525 }
1526
1527 #region From Triplets
1528
1529 //public static void DeleteSequence(Link sequence)
1530 //{
1531 //}
1532
1533 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1534 public List<ulong> CollectMatchingSequences(ulong[] links)
1535 {
1536     if (links.Length == 1)
1537     {

```

```

1537         throw new InvalidOperationException("Подпоследовательности с одним элементом не
1538             ↳ поддерживаются.");
1539     }
1540     var leftBound = 0;
1541     var rightBound = links.Length - 1;
1542     var left = links[leftBound++];
1543     var right = links[rightBound--];
1544     var results = new List<ulong>();
1545     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1546     return results;
1547 }
1548 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1549 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1550     ↳ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1551 {
1552     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1553     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1554     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1555     {
1556         var nextLeftLink = middleLinks[leftBound];
1557         var elements = GetRightElements(leftLink, nextLeftLink);
1558         if (leftBound <= rightBound)
1559         {
1560             for (var i = elements.Length - 1; i >= 0; i--)
1561             {
1562                 var element = elements[i];
1563                 if (element != 0)
1564                 {
1565                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
1566                         ↳ rightLink, rightBound, ref results);
1567                 }
1568             }
1569         }
1570         else
1571         {
1572             for (var i = elements.Length - 1; i >= 0; i--)
1573             {
1574                 var element = elements[i];
1575                 if (element != 0)
1576                 {
1577                     results.Add(element);
1578                 }
1579             }
1580         }
1581     }
1582     else
1583     {
1584         var nextRightLink = middleLinks[rightBound];
1585         var elements = GetLeftElements(rightLink, nextRightLink);
1586         if (leftBound <= rightBound)
1587         {
1588             for (var i = elements.Length - 1; i >= 0; i--)
1589             {
1590                 var element = elements[i];
1591                 if (element != 0)
1592                 {
1593                     CollectMatchingSequences(leftLink, leftBound, middleLinks,
1594                         ↳ elements[i], rightBound - 1, ref results);
1595                 }
1596             }
1597         }
1598         else
1599         {
1600             for (var i = elements.Length - 1; i >= 0; i--)
1601             {
1602                 var element = elements[i];
1603                 if (element != 0)
1604                 {
1605                     results.Add(element);
1606                 }
1607             }
1608         }
1609     }
1610 }
1611 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1612 public ulong[] GetRightElements(ulong startLink, ulong rightLink)

```

```

1611 {
1612     var result = new ulong[5];
1613     TryStepRight(startLink, rightLink, result, 0);
1614     Links.Each(Constants.Any, startLink, couple =>
1615     {
1616         if (couple != startLink)
1617         {
1618             if (TryStepRight(couple, rightLink, result, 2))
1619             {
1620                 return false;
1621             }
1622         }
1623         return true;
1624     });
1625     if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1626     {
1627         result[4] = startLink;
1628     }
1629     return result;
1630 }
1631
1632 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1633 public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1634 {
1635     var added = 0;
1636     Links.Each(startLink, Constants.Any, couple =>
1637     {
1638         if (couple != startLink)
1639         {
1640             var coupleTarget = Links.GetTarget(couple);
1641             if (coupleTarget == rightLink)
1642             {
1643                 result[offset] = couple;
1644                 if (++added == 2)
1645                 {
1646                     return false;
1647                 }
1648             }
1649             else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
1650                 ↪ == Net.And &&
1651             {
1652                 result[offset + 1] = couple;
1653                 if (++added == 2)
1654                 {
1655                     return false;
1656                 }
1657             }
1658         }
1659         return true;
1660     });
1661     return added > 0;
1662 }
1663
1664 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1665 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1666 {
1667     var result = new ulong[5];
1668     TryStepLeft(startLink, leftLink, result, 0);
1669     Links.Each(startLink, Constants.Any, couple =>
1670     {
1671         if (couple != startLink)
1672         {
1673             if (TryStepLeft(couple, leftLink, result, 2))
1674             {
1675                 return false;
1676             }
1677         }
1678         return true;
1679     });
1680     if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1681     {
1682         result[4] = leftLink;
1683     }
1684     return result;
1685 }
1686
1687 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1688 public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1689 {

```

```

1689     var added = 0;
1690     Links.Each(Constants.Any, startLink, couple =>
1691     {
1692         if (couple != startLink)
1693         {
1694             var coupleSource = Links.GetSource(couple);
1695             if (coupleSource == leftLink)
1696             {
1697                 result[offset] = couple;
1698                 if (++added == 2)
1699                 {
1700                     return false;
1701                 }
1702             }
1703             else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1704                 ⇨ == Net.And &&
1705             {
1706                 result[offset + 1] = couple;
1707                 if (++added == 2)
1708                 {
1709                     return false;
1710                 }
1711             }
1712             return true;
1713         });
1714         return added > 0;
1715     }
1716
1717 #endregion
1718
1719 #region Walkers
1720
1721 public class PatternMatcher : RightSequenceWalker<ulong>
1722 {
1723     private readonly Sequences _sequences;
1724     private readonly ulong[] _patternSequence;
1725     private readonly HashSet<LinkIndex> _linksInSequence;
1726     private readonly HashSet<LinkIndex> _results;
1727
1728     #region Pattern Match
1729
1730     enum PatternBlockType
1731     {
1732         Undefined,
1733         Gap,
1734         Elements
1735     }
1736
1737     struct PatternBlock
1738     {
1739         public PatternBlockType Type;
1740         public long Start;
1741         public long Stop;
1742     }
1743
1744     private readonly List<PatternBlock> _pattern;
1745     private int _patternPosition;
1746     private long _sequencePosition;
1747
1748     #endregion
1749
1750     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1751     public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1752         ⇨ HashSet<LinkIndex> results)
1753         : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1754     {
1755         _sequences = sequences;
1756         _patternSequence = patternSequence;
1757         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1758             ⇨ _sequences.Constants.Any && x != ZeroOrMany));
1759         _results = results;
1760         _pattern = CreateDetailedPattern();
1761     }
1762
1763     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1764     protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
1765         ⇨ base.IsElement(link);
1766
1767     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

1765 public bool PatternMatch(LinkIndex sequenceToMatch)
1766 {
1767     _patternPosition = 0;
1768     _sequencePosition = 0;
1769     foreach (var part in Walk(sequenceToMatch))
1770     {
1771         if (!PatternMatchCore(part))
1772         {
1773             break;
1774         }
1775     }
1776     return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1777         ↪ - 1 && _pattern[_patternPosition].Start == 0);
1778 }
1779 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1780 private List<PatternBlock> CreateDetailedPattern()
1781 {
1782     var pattern = new List<PatternBlock>();
1783     var patternBlock = new PatternBlock();
1784     for (var i = 0; i < _patternSequence.Length; i++)
1785     {
1786         if (patternBlock.Type == PatternBlockType.Undefined)
1787         {
1788             if (_patternSequence[i] == _sequences.Constants.Any)
1789             {
1790                 patternBlock.Type = PatternBlockType.Gap;
1791                 patternBlock.Start = 1;
1792                 patternBlock.Stop = 1;
1793             }
1794             else if (_patternSequence[i] == ZeroOrMany)
1795             {
1796                 patternBlock.Type = PatternBlockType.Gap;
1797                 patternBlock.Start = 0;
1798                 patternBlock.Stop = long.MaxValue;
1799             }
1800             else
1801             {
1802                 patternBlock.Type = PatternBlockType.Elements;
1803                 patternBlock.Start = i;
1804                 patternBlock.Stop = i;
1805             }
1806         }
1807         else if (patternBlock.Type == PatternBlockType.Elements)
1808         {
1809             if (_patternSequence[i] == _sequences.Constants.Any)
1810             {
1811                 pattern.Add(patternBlock);
1812                 patternBlock = new PatternBlock
1813                 {
1814                     Type = PatternBlockType.Gap,
1815                     Start = 1,
1816                     Stop = 1
1817                 };
1818             }
1819             else if (_patternSequence[i] == ZeroOrMany)
1820             {
1821                 pattern.Add(patternBlock);
1822                 patternBlock = new PatternBlock
1823                 {
1824                     Type = PatternBlockType.Gap,
1825                     Start = 0,
1826                     Stop = long.MaxValue
1827                 };
1828             }
1829             else
1830             {
1831                 patternBlock.Stop = i;
1832             }
1833         }
1834         else // patternBlock.Type == PatternBlockType.Gap
1835         {
1836             if (_patternSequence[i] == _sequences.Constants.Any)
1837             {
1838                 patternBlock.Start++;
1839                 if (patternBlock.Stop < patternBlock.Start)
1840                 {
1841                     patternBlock.Stop = patternBlock.Start;
1842                 }
1843             }
1844         }
1845     }

```

```

1844         else if (_patternSequence[i] == ZeroOrMany)
1845         {
1846             patternBlock.Stop = long.MaxValue;
1847         }
1848         else
1849         {
1850             pattern.Add(patternBlock);
1851             patternBlock = new PatternBlock
1852             {
1853                 Type = PatternBlockType.Elements,
1854                 Start = i,
1855                 Stop = i
1856             };
1857         }
1858     }
1859 }
1860 if (patternBlock.Type != PatternBlockType.Undefined)
1861 {
1862     pattern.Add(patternBlock);
1863 }
1864 return pattern;
1865 }
1866
1867 // match: search for regexp anywhere in text
1868 //int match(char* regexp, char* text)
1869 //{
1870 //    do
1871 //    {
1872 //        } while (*text++ != '\0');
1873 //    return 0;
1874 //}
1875
1876 // matchhere: search for regexp at beginning of text
1877 //int matchhere(char* regexp, char* text)
1878 //{
1879 //    if (regexp[0] == '\0')
1880 //        return 1;
1881 //    if (regexp[1] == '*')
1882 //        return matchstar(regexp[0], regexp + 2, text);
1883 //    if (regexp[0] == '$' && regexp[1] == '\0')
1884 //        return *text == '\0';
1885 //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1886 //        return matchhere(regexp + 1, text + 1);
1887 //    return 0;
1888 //}
1889
1890 // matchstar: search for c*regexp at beginning of text
1891 //int matchstar(int c, char* regexp, char* text)
1892 //{
1893 //    do
1894 //    {
1895 //        /* a * matches zero or more instances */
1896 //        if (matchhere(regexp, text))
1897 //            return 1;
1898 //    } while (*text != '\0' && (*text++ == c || c == '.'));
1899 //    return 0;
1900 //}
1901
1902 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
1903 //    ↪ long maximumGap)
1904 //{
1905 //    mininumGap = 0;
1906 //    maximumGap = 0;
1907 //    element = 0;
1908 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1909 //    {
1910 //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1911 //            mininumGap++;
1912 //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1913 //            maximumGap = long.MaxValue;
1914 //        else
1915 //            break;
1916 //    }
1917
1918 //    if (maximumGap < mininumGap)
1919 //        maximumGap = mininumGap;
1920 //}
1921
1922 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1923 private bool PatternMatchCore(LinkIndex element)

```

```

1922 {
1923     if (_patternPosition >= _pattern.Count)
1924     {
1925         _patternPosition = -2;
1926         return false;
1927     }
1928     var currentPatternBlock = _pattern[_patternPosition];
1929     if (currentPatternBlock.Type == PatternBlockType.Gap)
1930     {
1931         //var currentMatchingBlockLength = (_sequencePosition -
1932         ↪ _lastMatchedBlockPosition);
1933         if (_sequencePosition < currentPatternBlock.Start)
1934         {
1935             _sequencePosition++;
1936             return true; // Двигаемся дальше
1937         }
1938         // Это последний блок
1939         if (_pattern.Count == _patternPosition + 1)
1940         {
1941             _patternPosition++;
1942             _sequencePosition = 0;
1943             return false; // Полное соответствие
1944         }
1945         else
1946         {
1947             if (_sequencePosition > currentPatternBlock.Stop)
1948             {
1949                 return false; // Соответствие невозможно
1950             }
1951             var nextPatternBlock = _pattern[_patternPosition + 1];
1952             if (_patternSequence[nextPatternBlock.Start] == element)
1953             {
1954                 if (nextPatternBlock.Start < nextPatternBlock.Stop)
1955                 {
1956                     _patternPosition++;
1957                     _sequencePosition = 1;
1958                 }
1959                 else
1960                 {
1961                     _patternPosition += 2;
1962                     _sequencePosition = 0;
1963                 }
1964             }
1965         }
1966     }
1967     else // currentPatternBlock.Type == PatternBlockType.Elements
1968     {
1969         var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1970         if (_patternSequence[patternElementPosition] != element)
1971         {
1972             return false; // Соответствие невозможно
1973         }
1974         if (patternElementPosition == currentPatternBlock.Stop)
1975         {
1976             _patternPosition++;
1977             _sequencePosition = 0;
1978         }
1979         else
1980         {
1981             _sequencePosition++;
1982         }
1983     }
1984     return true;
1985     //if (_patternSequence[_patternPosition] != element)
1986     //    return false;
1987     //else
1988     //{
1989     //    _sequencePosition++;
1990     //    _patternPosition++;
1991     //    return true;
1992     //}
1993     //if (_filterPosition == _patternSequence.Length)
1994     //{
1995     //    _filterPosition = -2; // Длиннее чем нужно
1996     //    return false;
1997     //}
1998     //if (element != _patternSequence[_filterPosition])
1999     //{

```

```

2000         // _filterPosition = -1;
2001         // return false; // Начинается иначе
2002     //}
2003     //_filterPosition++;
2004     //if (_filterPosition == (_patternSequence.Length - 1))
2005     //    return false;
2006     //if (_filterPosition >= 0)
2007     //{
2008     //    if (element == _patternSequence[_filterPosition + 1])
2009     //        _filterPosition++;
2010     //    else
2011     //        return false;
2012     //}
2013     //if (_filterPosition < 0)
2014     //{
2015     //    if (element == _patternSequence[0])
2016     //        _filterPosition = 0;
2017     //}
2018 }
2019
2020 [MethodImpl(MethodImplOptions.AggressiveInlining)]
2021 public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
2022 {
2023     foreach (var sequenceToMatch in sequencesToMatch)
2024     {
2025         if (PatternMatch(sequenceToMatch))
2026         {
2027             _results.Add(sequenceToMatch);
2028         }
2029     }
2030 }
2031 }
2032
2033 #endregion
2034 }
2035 }

```

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

```

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

```

```

38     /// Писать тесты.
39     ///
40     ///
41     /// Можно убрать зависимость от конкретной реализации Links,
42     /// на зависимость от абстрактного элемента, который может быть представлен несколькими
43     /// ↪ способами.
44     ///
45     /// Можно ли как-то сделать один общий интерфейс
46     ///
47     /// Блокчейн и/или гит для распределённой записи транзакций.
48     ///
49     /// </remarks>
50     public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
51     ↪ (после завершения реализации Sequences)
52     {
53         /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
54         ↪ связей.</summary>
55         public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
56
57         public SequencesOptions<LinkIndex> Options { get; }
58         public SynchronizedLinks<LinkIndex> Links { get; }
59         private readonly ISynchronization _sync;
60
61         public LinksConstants<LinkIndex> Constants { get; }
62
63         [MethodImpl(MethodImplOptions.AggressiveInlining)]
64         public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
65         {
66             Links = links;
67             _sync = links.SyncRoot;
68             Options = options;
69             Options.ValidateOptions();
70             Options.InitOptions(Links);
71             Constants = links.Constants;
72         }
73
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]
75         public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
76         ↪ SequencesOptions<LinkIndex>()) { }
77
78         [MethodImpl(MethodImplOptions.AggressiveInlining)]
79         public bool IsSequence(LinkIndex sequence)
80         {
81             return _sync.ExecuteReadOperation(() =>
82             {
83                 if (Options.UseSequenceMarker)
84                 {
85                     return Options.MarkedSequenceMatcher.IsMatched(sequence);
86                 }
87                 return !Links.Unsync.IsPartialPoint(sequence);
88             });
89         }
90
91         [MethodImpl(MethodImplOptions.AggressiveInlining)]
92         private LinkIndex GetSequenceByElements(LinkIndex sequence)
93         {
94             if (Options.UseSequenceMarker)
95             {
96                 return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
97             }
98             return sequence;
99         }
100
101         [MethodImpl(MethodImplOptions.AggressiveInlining)]
102         private LinkIndex GetSequenceElements(LinkIndex sequence)
103         {
104             if (Options.UseSequenceMarker)
105             {
106                 var linkContents = new Link<ulong>(Links.GetLink(sequence));
107                 if (linkContents.Source == Options.SequenceMarkerLink)
108                 {
109                     return linkContents.Target;
110                 }
111                 if (linkContents.Target == Options.SequenceMarkerLink)
112                 {
113                     return linkContents.Source;
114                 }
115             }
116         }

```

```

113     return sequence;
114 }
115
116 #region Count
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 public LinkIndex Count(ICollection<LinkIndex> restrictions)
120 {
121     if (restrictions.IsNullOrEmpty())
122     {
123         return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
124     }
125     if (restrictions.Count == 1) // Первая связь это адрес
126     {
127         var sequenceIndex = restrictions[0];
128         if (sequenceIndex == Constants.Null)
129         {
130             return 0;
131         }
132         if (sequenceIndex == Constants.Any)
133         {
134             return Count(null);
135         }
136         if (Options.UseSequenceMarker)
137         {
138             return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
139         }
140         return Links.Exists(sequenceIndex) ? 1UL : 0;
141     }
142     throw new NotImplementedException();
143 }
144
145 [MethodImpl(MethodImplOptions.AggressiveInlining)]
146 private LinkIndex CountUsages(params LinkIndex[] restrictions)
147 {
148     if (restrictions.Length == 0)
149     {
150         return 0;
151     }
152     if (restrictions.Length == 1) // Первая связь это адрес
153     {
154         if (restrictions[0] == Constants.Null)
155         {
156             return 0;
157         }
158         var any = Constants.Any;
159         if (Options.UseSequenceMarker)
160         {
161             var elementsLink = GetSequenceElements(restrictions[0]);
162             var sequenceLink = GetSequenceByElements(elementsLink);
163             if (sequenceLink != Constants.Null)
164             {
165                 return Links.Count(any, sequenceLink) + Links.Count(any, elementsLink) -
166                     ↪ 1;
167             }
168             return Links.Count(any, elementsLink);
169         }
170         return Links.Count(any, restrictions[0]);
171     }
172     throw new NotImplementedException();
173 }
174
175 #endregion
176
177 #region Create
178
179 [MethodImpl(MethodImplOptions.AggressiveInlining)]
180 public LinkIndex Create(ICollection<LinkIndex> restrictions)
181 {
182     return _sync.ExecuteWriteOperation(() =>
183     {
184         if (restrictions.IsNullOrEmpty())
185         {
186             return Constants.Null;
187         }
188         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
189         return CreateCore(restrictions);
190     });
191 }

```

```

191 [MethodImpl(MethodImplOptions.AggressiveInlining)]
192 private LinkIndex CreateCore(ICollection<LinkIndex> restrictions)
193 {
194     LinkIndex[] sequence = restrictions.SkipFirst();
195     if (Options.UseIndex)
196     {
197         Options.Index.Add(sequence);
198     }
199     var sequenceRoot = default(LinkIndex);
200     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
201     {
202         var matches = Each(restrictions);
203         if (matches.Count > 0)
204         {
205             sequenceRoot = matches[0];
206         }
207     }
208     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
209     {
210         return CompactCore(sequence);
211     }
212     if (sequenceRoot == default)
213     {
214         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
215     }
216     if (Options.UseSequenceMarker)
217     {
218         return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
219     }
220     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
221 }
222
223 #endregion
224
225 #region Each
226
227 [MethodImpl(MethodImplOptions.AggressiveInlining)]
228 public List<LinkIndex> Each(ICollection<LinkIndex> sequence)
229 {
230     var results = new List<LinkIndex>();
231     var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
232     Each(filler.AddFirstAndReturnConstant, sequence);
233     return results;
234 }
235
236 [MethodImpl(MethodImplOptions.AggressiveInlining)]
237 public LinkIndex Each(Func<ICollection<LinkIndex>, LinkIndex> handler, ICollection<LinkIndex>
238     → restrictions)
239 {
240     return _sync.ExecuteReadOperation(() =>
241     {
242         if (restrictions.IsNullOrEmpty())
243         {
244             return Constants.Continue;
245         }
246         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
247         if (restrictions.Count == 1)
248         {
249             var link = restrictions[0];
250             var any = Constants.Any;
251             if (link == any)
252             {
253                 if (Options.UseSequenceMarker)
254                 {
255                     return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
256                         → Options.SequenceMarkerLink, any));
257                 }
258                 else
259                 {
260                     return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
261                         → any));
262                 }
263             }
264             if (Options.UseSequenceMarker)
265             {
266                 var sequenceLinkValues = Links.Unsync.GetLink(link);

```

```

265         if (sequenceLinkValues[Constants.SourcePart] ==
266             ↪ Options.SequenceMarkerLink)
267         {
268             link = sequenceLinkValues[Constants.TargetPart];
269         }
270         var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
271         sequence[0] = link;
272         return handler(sequence);
273     }
274     else if (restrictions.Count == 2)
275     {
276         throw new NotImplementedException();
277     }
278     else if (restrictions.Count == 3)
279     {
280         return Links.Unsync.Each(handler, restrictions);
281     }
282     else
283     {
284         var sequence = restrictions.SkipFirst();
285         if (Options.UseIndex && !Options.Index.MightContain(sequence))
286         {
287             return Constants.Break;
288         }
289         return EachCore(handler, sequence);
290     }
291     });
292 }
293
294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
296 ↪ values)
297 {
298     var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
299     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
300     ↪ Id.
301     Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
302     ↪ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
303     ↪ matcher.HandleFullMatched;
304     //if (sequence.Length >= 2)
305     if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
306     {
307         return Constants.Break;
308     }
309     var last = values.Count - 2;
310     for (var i = 1; i < last; i++)
311     {
312         if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
313             ↪ Constants.Continue)
314         {
315             return Constants.Break;
316         }
317     }
318     if (values.Count >= 3)
319     {
320         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
321             ↪ != Constants.Continue)
322         {
323             return Constants.Break;
324         }
325     }
326     return Constants.Continue;
327 }
328
329 [MethodImpl(MethodImplOptions.AggressiveInlining)]
330 private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
331 ↪ left, LinkIndex right)
332 {
333     return Links.Unsync.Each(doublet =>
334     {
335         var doubletIndex = doublet[Constants.IndexPart];
336         if (StepRight(handler, doubletIndex, right) != Constants.Continue)
337         {
338             return Constants.Break;
339         }
340         if (left != doubletIndex)
341         {

```



```

335         return PartialStepRight(handler, doubletIndex, right);
336     }
337     return Constants.Continue;
338 }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
339 }
340
341 [MethodImpl(MethodImplOptions.AggressiveInlining)]
342 private LinkIndex StepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↳ LinkIndex right) => Links.Unsync.Each(rightStep => TryStepRightUp(handler, right,
    ↳ rightStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, left,
    ↳ Constants.Any));
343
344 [MethodImpl(MethodImplOptions.AggressiveInlining)]
345 private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ right, LinkIndex stepFrom)
346 {
347     var upStep = stepFrom;
348     var firstSource = Links.Unsync.GetTarget(upStep);
349     while (firstSource != right && firstSource != upStep)
350     {
351         upStep = firstSource;
352         firstSource = Links.Unsync.GetSource(upStep);
353     }
354     if (firstSource == right)
355     {
356         return handler(new LinkAddress<LinkIndex>(stepFrom));
357     }
358     return Constants.Continue;
359 }
360
361 [MethodImpl(MethodImplOptions.AggressiveInlining)]
362 private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↳ LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
    ↳ leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
    ↳ right));
363
364 [MethodImpl(MethodImplOptions.AggressiveInlining)]
365 private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ left, LinkIndex stepFrom)
366 {
367     var upStep = stepFrom;
368     var firstTarget = Links.Unsync.GetSource(upStep);
369     while (firstTarget != left && firstTarget != upStep)
370     {
371         upStep = firstTarget;
372         firstTarget = Links.Unsync.GetTarget(upStep);
373     }
374     if (firstTarget == left)
375     {
376         return handler(new LinkAddress<LinkIndex>(stepFrom));
377     }
378     return Constants.Continue;
379 }
380
381 #endregion
382
383 #region Update
384
385 [MethodImpl(MethodImplOptions.AggressiveInlining)]
386 public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
387 {
388     var sequence = restrictions.SkipFirst();
389     var newSequence = substitution.SkipFirst();
390     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
391     {
392         return Constants.Null;
393     }
394     if (sequence.IsNullOrEmpty())
395     {
396         return Create(substitution);
397     }
398     if (newSequence.IsNullOrEmpty())
399     {
400         Delete(restrictions);
401         return Constants.Null;
402     }
403     return _sync.ExecuteWriteOperation((Func<ulong>)(() =>
404     {
405         ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);

```

```

406         Links.EnsureLinkExists(newSequence);
407         return UpdateCore(sequence, newSequence);
408     }));
409 }
410
411 [MethodImpl(MethodImplOptions.AggressiveInlining)]
412 private LinkIndex UpdateCore(ICollection<LinkIndex> sequence, ICollection<LinkIndex> newSequence)
413 {
414     LinkIndex bestVariant;
415     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
416         ↪ !sequence.EqualTo(newSequence))
417     {
418         bestVariant = CompactCore(newSequence);
419     }
420     else
421     {
422         bestVariant = CreateCore(newSequence);
423     }
424     // TODO: Check all options only ones before loop execution
425     // Возможно нужно две версии Each, возвращающий фактические последовательности и с
426     ↪ маркером,
427     // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
428     ↪ можно получить имея только фактические последовательности.
429     foreach (var variant in Each(sequence))
430     {
431         if (variant != bestVariant)
432         {
433             UpdateOneCore(variant, bestVariant);
434         }
435     }
436     return bestVariant;
437 }
438
439 [MethodImpl(MethodImplOptions.AggressiveInlining)]
440 private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
441 {
442     if (Options.UseGarbageCollection)
443     {
444         var sequenceElements = GetSequenceElements(sequence);
445         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
446         var sequenceLink = GetSequenceByElements(sequenceElements);
447         var newSequenceElements = GetSequenceElements(newSequence);
448         var newSequenceLink = GetSequenceByElements(newSequenceElements);
449         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
450         {
451             if (sequenceLink != Constants.Null)
452             {
453                 Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
454             }
455             Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
456         }
457         ClearGarbage(sequenceElementsContents.Source);
458         ClearGarbage(sequenceElementsContents.Target);
459     }
460     else
461     {
462         if (Options.UseSequenceMarker)
463         {
464             var sequenceElements = GetSequenceElements(sequence);
465             var sequenceLink = GetSequenceByElements(sequenceElements);
466             var newSequenceElements = GetSequenceElements(newSequence);
467             var newSequenceLink = GetSequenceByElements(newSequenceElements);
468             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
469             {
470                 if (sequenceLink != Constants.Null)
471                 {
472                     Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
473                 }
474                 Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
475             }
476         }
477         else
478         {
479             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
480             {
481                 Links.Unsync.MergeAndDelete(sequence, newSequence);
482             }
483         }
484     }
485 }

```

```

    }
}

#endregion

#region Delete

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

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

#endregion

#region Compactification

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void CompactAll()
{
    _sync.ExecuteWriteOperation(() =>
    {
        var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
        for (int i = 0; i < sequences.Count; i++)
        {
            var sequence = this.ToList(sequences[i]);
            Compact(sequence.ShiftRight());
        }
    });
}

#endregion

```

```

560     }
561     });
562 }
563
564 /// <remarks>
565 /// bestVariant можно выбирать по максимальному числу использований,
566 /// но балансированный позволяет гарантировать уникальность (если есть возможность,
567 /// гарантировать его использование в других местах).
568 ///
569 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
570 /// </remarks>
571 [MethodImpl(MethodImplOptions.AggressiveInlining)]
572 public LinkIndex Compact(ICollection<LinkIndex> sequence)
573 {
574     return _sync.ExecuteWriteOperation(() =>
575     {
576         if (sequence.IsNullOrEmpty())
577         {
578             return Constants.Null;
579         }
580         Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
581         return CompactCore(sequence);
582     });
583 }
584
585 [MethodImpl(MethodImplOptions.AggressiveInlining)]
586 private LinkIndex CompactCore(ICollection<LinkIndex> sequence) => UpdateCore(sequence,
587     ↪ sequence);
588
589 #endregion
590
591 #region Garbage Collection
592
593 /// <remarks>
594 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
595 ↪ определить извне или в унаследованном классе
596 /// </remarks>
597 [MethodImpl(MethodImplOptions.AggressiveInlining)]
598 private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
599     ↪ !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
600
601 [MethodImpl(MethodImplOptions.AggressiveInlining)]
602 private void ClearGarbage(LinkIndex link)
603 {
604     if (IsGarbage(link))
605     {
606         var contents = new Link<ulong>(Links.GetLink(link));
607         Links.Unsync.Delete(link);
608         ClearGarbage(contents.Source);
609         ClearGarbage(contents.Target);
610     }
611 }
612
613 #endregion
614
615 #region Walkers
616
617 [MethodImpl(MethodImplOptions.AggressiveInlining)]
618 public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
619 {
620     return _sync.ExecuteReadOperation(() =>
621     {
622         var links = Links.Unsync;
623         foreach (var part in Options.Walker.Walk(sequence))
624         {
625             if (!handler(part))
626             {
627                 return false;
628             }
629         }
630         return true;
631     });
632 }
633
634 public class Matcher : RightSequenceWalker<LinkIndex>
635 {
636     private readonly Sequences _sequences;
637     private readonly ICollection<LinkIndex> _patternSequence;
638     private readonly HashSet<LinkIndex> _linksInSequence;

```

```

636 private readonly HashSet<LinkIndex> _results;
637 private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
638 private readonly HashSet<LinkIndex> _readAsElements;
639 private int _filterPosition;
640
641 [MethodImpl(MethodImplOptions.AggressiveInlining)]
642 public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
    ↳ HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
    ↳ HashSet<LinkIndex> readAsElements = null)
    : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
643 {
644     _sequences = sequences;
645     _patternSequence = patternSequence;
646     _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
    ↳ _links.Constants.Any && x != ZeroOrMany));
647     _results = results;
648     _stopableHandler = stopableHandler;
649     _readAsElements = readAsElements;
650 }
651
652 [MethodImpl(MethodImplOptions.AggressiveInlining)]
653 protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
    ↳ (_readAsElements != null && _readAsElements.Contains(link)) ||
    ↳ _linksInSequence.Contains(link);
654
655 [MethodImpl(MethodImplOptions.AggressiveInlining)]
656 public bool FullMatch(LinkIndex sequenceToMatch)
657 {
658     _filterPosition = 0;
659     foreach (var part in Walk(sequenceToMatch))
660     {
661         if (!FullMatchCore(part))
662         {
663             break;
664         }
665     }
666     return _filterPosition == _patternSequence.Count;
667 }
668
669 [MethodImpl(MethodImplOptions.AggressiveInlining)]
670 private bool FullMatchCore(LinkIndex element)
671 {
672     if (_filterPosition == _patternSequence.Count)
673     {
674         _filterPosition = -2; // Длиннее чем нужно
675         return false;
676     }
677     if (_patternSequence[_filterPosition] != _links.Constants.Any
    ↳ && element != _patternSequence[_filterPosition])
678     {
679         _filterPosition = -1;
680         return false; // Начинается/Продолжается иначе
681     }
682     _filterPosition++;
683     return true;
684 }
685
686 [MethodImpl(MethodImplOptions.AggressiveInlining)]
687 public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
688 {
689     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
690     if (FullMatch(sequenceToMatch))
691     {
692         _results.Add(sequenceToMatch);
693     }
694 }
695
696 [MethodImpl(MethodImplOptions.AggressiveInlining)]
697 public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
698 {
699     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
700     if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
701     {
702         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
703     }
704     return _links.Constants.Continue;
705 }
706
707 [MethodImpl(MethodImplOptions.AggressiveInlining)]
708
709

```

```

710 public LinkIndex HandleFullMatchedSequence(IList<LinkIndex> restrictions)
711 {
712     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
713     var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
714     if (sequence != _links.Constants.Null && FullMatch(sequenceToMatch) &&
        ↪ _results.Add(sequenceToMatch))
715     {
716         return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
717     }
718     return _links.Constants.Continue;
719 }
720
721 /// <remarks>
722 /// TODO: Add support for LinksConstants.Any
723 /// </remarks>
724 [MethodImpl(MethodImplOptions.AggressiveInlining)]
725 public bool PartialMatch(LinkIndex sequenceToMatch)
726 {
727     _filterPosition = -1;
728     foreach (var part in Walk(sequenceToMatch))
729     {
730         if (!PartialMatchCore(part))
731         {
732             break;
733         }
734     }
735     return _filterPosition == _patternSequence.Count - 1;
736 }
737
738 [MethodImpl(MethodImplOptions.AggressiveInlining)]
739 private bool PartialMatchCore(LinkIndex element)
740 {
741     if (_filterPosition == (_patternSequence.Count - 1))
742     {
743         return false; // Нашлось
744     }
745     if (_filterPosition >= 0)
746     {
747         if (element == _patternSequence[_filterPosition + 1])
748         {
749             _filterPosition++;
750         }
751         else
752         {
753             _filterPosition = -1;
754         }
755     }
756     if (_filterPosition < 0)
757     {
758         if (element == _patternSequence[0])
759         {
760             _filterPosition = 0;
761         }
762     }
763     return true; // Ищем дальше
764 }
765
766 [MethodImpl(MethodImplOptions.AggressiveInlining)]
767 public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
768 {
769     if (PartialMatch(sequenceToMatch))
770     {
771         _results.Add(sequenceToMatch);
772     }
773 }
774
775 [MethodImpl(MethodImplOptions.AggressiveInlining)]
776 public LinkIndex HandlePartialMatched(IList<LinkIndex> restrictions)
777 {
778     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
779     if (PartialMatch(sequenceToMatch))
780     {
781         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
782     }
783     return _links.Constants.Continue;
784 }
785
786 [MethodImpl(MethodImplOptions.AggressiveInlining)]
787 public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)

```

```

788     {
789         foreach (var sequenceToMatch in sequencesToMatch)
790         {
791             if (PartialMatch(sequenceToMatch))
792             {
793                 _results.Add(sequenceToMatch);
794             }
795         }
796     }
797
798     [MethodImpl(MethodImplOptions.AggressiveInlining)]
799     public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
800     ↪ sequencesToMatch)
801     {
802         foreach (var sequenceToMatch in sequencesToMatch)
803         {
804             if (PartialMatch(sequenceToMatch))
805             {
806                 _readAsElements.Add(sequenceToMatch);
807                 _results.Add(sequenceToMatch);
808             }
809         }
810     }
811     #endregion
812 }
813
814 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences
8  {
9      public static class SequencesExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static TLink Create<TLink>(this ILinks<TLink> sequences, IList<TLink[]>
13         ↪ groupedSequence)
14         {
15             var finalSequence = new TLink[groupedSequence.Count];
16             for (var i = 0; i < finalSequence.Length; i++)
17             {
18                 var part = groupedSequence[i];
19                 finalSequence[i] = part.Length == 1 ? part[0] :
20                 ↪ sequences.Create(part.ShiftRight());
21             }
22             return sequences.Create(finalSequence.ShiftRight());
23         }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
27         {
28             var list = new List<TLink>();
29             var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
30             sequences.Each(filler.AddSkipFirstAndReturnConstant, new
31             ↪ LinkAddress<TLink>(sequence));
32             return list;
33         }
34     }
35 }

```

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

```

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

```

```

13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
19     ↪ ILinks<TLink> must contain GetConstants function.
20     {
21         private static readonly EqualityComparer<TLink> _equalityComparer =
22         ↪ EqualityComparer<TLink>.Default;
23
24         public TLink SequenceMarkerLink
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28             [MethodImpl(MethodImplOptions.AggressiveInlining)]
29             set;
30         }
31
32         public bool UseCascadeUpdate
33         {
34             [MethodImpl(MethodImplOptions.AggressiveInlining)]
35             get;
36             [MethodImpl(MethodImplOptions.AggressiveInlining)]
37             set;
38         }
39
40         public bool UseCascadeDelete
41         {
42             [MethodImpl(MethodImplOptions.AggressiveInlining)]
43             get;
44             [MethodImpl(MethodImplOptions.AggressiveInlining)]
45             set;
46         }
47
48         public bool UseIndex
49         {
50             [MethodImpl(MethodImplOptions.AggressiveInlining)]
51             get;
52             [MethodImpl(MethodImplOptions.AggressiveInlining)]
53             set;
54         } // TODO: Update Index on sequence update/delete.
55
56         public bool UseSequenceMarker
57         {
58             [MethodImpl(MethodImplOptions.AggressiveInlining)]
59             get;
60             [MethodImpl(MethodImplOptions.AggressiveInlining)]
61             set;
62         }
63
64         public bool UseCompression
65         {
66             [MethodImpl(MethodImplOptions.AggressiveInlining)]
67             get;
68             [MethodImpl(MethodImplOptions.AggressiveInlining)]
69             set;
70         }
71
72         public bool UseGarbageCollection
73         {
74             [MethodImpl(MethodImplOptions.AggressiveInlining)]
75             get;
76             [MethodImpl(MethodImplOptions.AggressiveInlining)]
77             set;
78         }
79
80         public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting
81         {
82             [MethodImpl(MethodImplOptions.AggressiveInlining)]
83             get;
84             [MethodImpl(MethodImplOptions.AggressiveInlining)]
85             set;
86         }
87
88         public bool EnforceSingleSequenceVersionOnWriteBasedOnNew
89         {
90             [MethodImpl(MethodImplOptions.AggressiveInlining)]
91             get;
92             [MethodImpl(MethodImplOptions.AggressiveInlining)]
93             set;
94         }
95     }
96 }

```



```

92     }
93
94     public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher
95     {
96         [MethodImpl(MethodImplOptions.AggressiveInlining)]
97         get;
98         [MethodImpl(MethodImplOptions.AggressiveInlining)]
99         set;
100     }
101
102     public IConverter<IList<TLink>, TLink> LinksToSequenceConverter
103     {
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         get;
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         set;
108     }
109
110     public ISequenceIndex<TLink> Index
111     {
112         [MethodImpl(MethodImplOptions.AggressiveInlining)]
113         get;
114         [MethodImpl(MethodImplOptions.AggressiveInlining)]
115         set;
116     }
117
118     public ISequenceWalker<TLink> Walker
119     {
120         [MethodImpl(MethodImplOptions.AggressiveInlining)]
121         get;
122         [MethodImpl(MethodImplOptions.AggressiveInlining)]
123         set;
124     }
125
126     public bool ReadFullSequence
127     {
128         [MethodImpl(MethodImplOptions.AggressiveInlining)]
129         get;
130         [MethodImpl(MethodImplOptions.AggressiveInlining)]
131         set;
132     }
133
134     // TODO: Реализовать компактификацию при чтении
135     //public bool EnforceSingleSequenceVersionOnRead { get; set; }
136     //public bool UseRequestMarker { get; set; }
137     //public bool StoreRequestResults { get; set; }
138
139     [MethodImpl(MethodImplOptions.AggressiveInlining)]
140     public void InitOptions(ISynchronizedLinks<TLink> links)
141     {
142         if (UseSequenceMarker)
143         {
144             if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
145             {
146                 SequenceMarkerLink = links.CreatePoint();
147             }
148             else
149             {
150                 if (!links.Exists(SequenceMarkerLink))
151                 {
152                     var link = links.CreatePoint();
153                     if (!_equalityComparer.Equals(link, SequenceMarkerLink))
154                     {
155                         throw new InvalidOperationException("Cannot recreate sequence marker
156                             ↪ link.");
157                     }
158                 }
159             }
160             if (MarkedSequenceMatcher == null)
161             {
162                 MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
163                     ↪ SequenceMarkerLink);
164             }
165         }
166         var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
167         if (UseCompression)
168         {
169             if (LinksToSequenceConverter == null)
170             {

```

```

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

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Walkers
7 {
8     public interface ISequenceWalker<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         IEnumerable<TLink> Walk(TLink sequence);
12     }
13 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Collections.Stacks;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Sequences.Walkers
9 {
10     public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
                ↪ isElement) : base(links, stack, isElement) { }
14

```

```

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

```

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

```

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

```

```

47 #endif
48         hasElements = false;
49         for (var i = 0; i < array.Length; i++)
50         {
51             var candidate = array[i];
52             if (!_equalityComparer.Equals(array[i], default))
53             {
54                 continue;
55             }
56             var doubletOffset = i * 2;
57             if (!_isElement(candidate))
58             {
59                 nextArray[doubletOffset] = candidate;
60             }
61             else
62             {
63                 var links = _links;
64                 var link = links.GetLink(candidate);
65                 var linkSource = links.GetSource(link);
66                 var linkTarget = links.GetTarget(link);
67                 nextArray[doubletOffset] = linkSource;
68                 nextArray[doubletOffset + 1] = linkTarget;
69                 if (!hasElements)
70                 {
71                     hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
72                 }
73             }
74         }
75 #if USEARRAYPOOL
76         if (array.Length > 1)
77         {
78             ArrayPool.Free(array);
79         }
80 #endif
81         array = nextArray;
82     }
83     while (hasElements);
84     var filledElementsCount = CountFilledElements(array);
85     if (filledElementsCount == array.Length)
86     {
87         return array;
88     }
89     else
90     {
91         return CopyFilledElements(array, filledElementsCount);
92     }
93 }
94
95 [MethodImpl(MethodImplOptions.AggressiveInlining)]
96 private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
97 {
98     var finalArray = new TLink[filledElementsCount];
99     for (int i = 0, j = 0; i < array.Length; i++)
100     {
101         if (!_equalityComparer.Equals(array[i], default))
102         {
103             finalArray[j] = array[i];
104             j++;
105         }
106     }
107 #if USEARRAYPOOL
108     ArrayPool.Free(array);
109 #endif
110     return finalArray;
111 }
112
113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
114 private static int CountFilledElements(TLink[] array)
115 {
116     var count = 0;
117     for (var i = 0; i < array.Length; i++)
118     {
119         if (!_equalityComparer.Equals(array[i], default))
120         {
121             count++;
122         }
123     }
124     return count;
125 }

```

```

126     }
127 }

```

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

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Stacks;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.Walkers
9  {
10     public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
11         ↪ ISequenceWalker<TLink>
12     {
13         private readonly IStack<TLink> _stack;
14         private readonly Func<TLink, bool> _isElement;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
18             ↪ isElement) : base(links)
19         {
20             _stack = stack;
21             _isElement = isElement;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
26             ↪ stack, links.IsPartialPoint) { }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public IEnumerable<TLink> Walk(TLink sequence)
30         {
31             _stack.Clear();
32         }
33     }
34 }

```

```

29     var element = sequence;
30     if (IsElement(element))
31     {
32         yield return element;
33     }
34     else
35     {
36         while (true)
37         {
38             if (IsElement(element))
39             {
40                 if (_stack.IsEmpty)
41                 {
42                     break;
43                 }
44                 element = _stack.Pop();
45                 foreach (var output in WalkContents(element))
46                 {
47                     yield return output;
48                 }
49                 element = GetNextElementAfterPop(element);
50             }
51             else
52             {
53                 _stack.Push(element);
54                 element = GetNextElementAfterPush(element);
55             }
56         }
57     }
58 }
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected abstract TLink GetNextElementAfterPop(TLink element);
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 protected abstract TLink GetNextElementAfterPush(TLink element);
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected abstract IEnumerable<TLink> WalkContents(TLink element);
71 }
72 }

```

1.161 ./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)]

```

```

34     public TLink Pop()
35     {
36         var element = Peek();
37         if (!_equalityComparer.Equals(element, _stack))
38         {
39             var top = GetTop();
40             var previousTop = _links.GetSource(top);
41             _links.Update(_stack, GetStackMarker(), previousTop);
42             _links.Delete(top);
43         }
44         return element;
45     }
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
49         ↪ _links.GetOrCreate(GetTop(), element));
50 }

```

1.162 ./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 }

```

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

```

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

```

```

42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
    ↳ ReaderWriterLockSynchronization(), links) { }
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
47 {
48     SyncRoot = synchronization;
49     Sync = this;
50     Unsync = links;
51     Constants = links.Constants;
52 }
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 public TLinkAddress Count(IList<TLinkAddress> restriction) =>
    ↳ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
    ↳ IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
    ↳ restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
    ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
    ↳ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
    ↳ Unsync.Update);
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 public void Delete(IList<TLinkAddress> restrictions) =>
    ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
68
69 //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
    ↳ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
70 //{
71 //    if (restriction != null && substitution != null &&
    ↳ !substitution.EqualTo(restriction))
72 //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
    ↳ substitution, substitutedHandler, Unsync.Trigger);
73
74 //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
    ↳ substitutedHandler, Unsync.Trigger);
75 //}
76 }
77 }

```

1.164 ./csharp/Platform.Data.Doublets/Time/DateTimeToLongRawNumberSequenceConverter.cs

```

1 using System;
2 using System.Runtime.CompilerServices;
3 using Platform.Converters;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Time
8 {
9     public class DateTimeToLongRawNumberSequenceConverter<TLink> : IConverter<DateTime, TLink>
10     {
11         private readonly IConverter<long, TLink> _int64ToLongRawNumberConverter;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public DateTimeToLongRawNumberSequenceConverter(IConverter<long, TLink>
            ↳ int64ToLongRawNumberConverter) => _int64ToLongRawNumberConverter =
            ↳ int64ToLongRawNumberConverter;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public TLink Convert(DateTime source) =>
            ↳ _int64ToLongRawNumberConverter.Convert(source.ToFileTimeUtc());
18     }
19 }

```

1.165 ./csharp/Platform.Data.Doublets/Time/LongRawNumberSequenceToDateTimeConverter.cs

```

1 using System;
2 using System.Runtime.CompilerServices;
3 using Platform.Converters;
4

```



```

5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Time
8 {
9     public class LongRawNumberSequenceToDateTimeConverter<TLink> : IConverter<TLink, DateTime>
10    {
11        private readonly IConverter<TLink, long> _longRawNumberConverterToInt64;
12
13        [MethodImpl(MethodImplOptions.AggressiveInlining)]
14        public LongRawNumberSequenceToDateTimeConverter(IConverter<TLink, long>
15            ↪ longRawNumberConverterToInt64) => _longRawNumberConverterToInt64 =
16            ↪ longRawNumberConverterToInt64;
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        public DateTime Convert(TLink source) =>
20            ↪ DateTime.FromFileTimeUtc(_longRawNumberConverterToInt64.Convert(source));
21    }
22 }

```

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

```

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

```

```

53 }
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
    ↳ HashSet<ulong> visited, ulong linkIndex, Func<Link<ulong>, bool> isElement,
    ↳ Action<StringBuilder, Link<ulong>> appendElement, bool renderIndex = false, bool
    ↳ renderDebug = false)
57 {
58     if (sb == null)
59     {
60         throw new ArgumentNullException(nameof(sb));
61     }
62     if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
    ↳ Constants.Itself)
63     {
64         return;
65     }
66     if (links.Exists(linkIndex))
67     {
68         if (visited.Add(linkIndex))
69         {
70             sb.Append('(');
71             var link = new Link<ulong>(links.GetLink(linkIndex));
72             if (renderIndex)
73             {
74                 sb.Append(link.Index);
75                 sb.Append(':');
76             }
77             if (link.Source == link.Index)
78             {
79                 sb.Append(link.Index);
80             }
81             else
82             {
83                 var source = new Link<ulong>(links.GetLink(link.Source));
84                 if (isElement(source))
85                 {
86                     appendElement(sb, source);
87                 }
88                 else
89                 {
90                     links.AppendStructure(sb, visited, source.Index, isElement,
    ↳ appendElement, renderIndex);
91                 }
92             }
93             sb.Append(' ');
94             if (link.Target == link.Index)
95             {
96                 sb.Append(link.Index);
97             }
98             else
99             {
100                 var target = new Link<ulong>(links.GetLink(link.Target));
101                 if (isElement(target))
102                 {
103                     appendElement(sb, target);
104                 }
105                 else
106                 {
107                     links.AppendStructure(sb, visited, target.Index, isElement,
    ↳ appendElement, renderIndex);
108                 }
109             }
110             sb.Append(')');
111         }
112         else
113         {
114             if (renderDebug)
115             {
116                 sb.Append('*');
117             }
118             sb.Append(linkIndex);
119         }
120     }
121     else
122     {
123         if (renderDebug)
124         {

```

```

125         sb.Append('~');
126     }
127     sb.Append(linkIndex);
128 }
129 }
130 }
131 }

```

1.167 ./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

```

```

71         throw new NotImplementedException();
72     }
73 }
74 }
75 ///
76 /// private struct Transition
77 /// {
78     public TransitionHeader Header;
79     public Link Source;
80     public Link Linker;
81     public Link Target;
82 }
83 ///
84 /// </remarks>
85 public struct Transition : IEquatable<Transition>
86 {
87     public static readonly long Size = Structure<Transition>.Size;
88
89     public readonly ulong TransactionId;
90     public readonly Link<ulong> Before;
91     public readonly Link<ulong> After;
92     public readonly Timestamp Timestamp;
93
94     [MethodImpl(MethodImplOptions.AggressiveInlining)]
95     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
96     ↪ transactionId, Link<ulong> before, Link<ulong> after)
97     {
98         TransactionId = transactionId;
99         Before = before;
100        After = after;
101        Timestamp = uniqueTimestampFactory.Create();
102    }
103
104     [MethodImpl(MethodImplOptions.AggressiveInlining)]
105     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
106     ↪ transactionId, Link<ulong> before) : this(uniqueTimestampFactory, transactionId,
107     ↪ before, default) { }
108
109     [MethodImpl(MethodImplOptions.AggressiveInlining)]
110     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
111     ↪ transactionId) : this(uniqueTimestampFactory, transactionId, default, default) {
112     ↪ }
113
114     [MethodImpl(MethodImplOptions.AggressiveInlining)]
115     public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
116     ↪ {After}";
117
118     [MethodImpl(MethodImplOptions.AggressiveInlining)]
119     public override bool Equals(object obj) => obj is Transition transition ?
120     ↪ Equals(transition) : false;
121
122     [MethodImpl(MethodImplOptions.AggressiveInlining)]
123     public override int GetHashCode() => (TransactionId, Before, After,
124     ↪ Timestamp).GetHashCode();
125
126     [MethodImpl(MethodImplOptions.AggressiveInlining)]
127     public bool Equals(Transition other) => TransactionId == other.TransactionId &&
128     ↪ Before == other.Before && After == other.After && Timestamp == other.Timestamp;
129
130     [MethodImpl(MethodImplOptions.AggressiveInlining)]
131     public static bool operator ==(Transition left, Transition right) =>
132     ↪ left.Equals(right);
133
134     [MethodImpl(MethodImplOptions.AggressiveInlining)]
135     public static bool operator !=(Transition left, Transition right) => !(left ==
136     ↪ right);
137 }
138
139 /// <remarks>
140 /// Другие варианты реализации транзакций (атомарности):
141 /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
142 ↪ Target)) и индексов.
143 /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
144 ↪ потребуется решить вопрос
145 /// со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
146 ↪ пересечениями идентификаторов.
147 ///
148 /// Где хранить промежуточный список транзакций?

```

```

135 ///
136 /// В оперативной памяти:
137 /// Минусы:
138 /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
139 /// так как нужно отдельно выделять память под список трансформаций.
140 /// 2. Выделенной оперативной памяти может не хватить, в том случае,
141 /// если транзакция использует слишком много трансформаций.
142 /// -> Можно использовать жёсткий диск для слишком длинных транзакций.
143 /// -> Максимальный размер списка трансформаций можно ограничить / задать
    ↪ константой.
144 /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
    ↪ создавая задержку.
145 ///
146 /// На жёстком диске:
147 /// Минусы:
148 /// 1. Длительный отклик, на запись каждой трансформации.
149 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
150 /// -> Это может решаться упаковкой/исключением дублирующих операций.
151 /// -> Также это может решаться тем, что короткие транзакции вообще
152 /// не будут записываться в случае отката.
153 /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
    ↪ операции (трансформации)
    ↪ будут записаны в лог.
154 ///
155 ///
156 </remarks>
157 public class Transaction : DisposableBase
158 {
159     private readonly Queue<Transition> _transitions;
160     private readonly UInt64LinksTransactionsLayer _layer;
161     public bool IsCommitted { get; private set; }
162     public bool IsReverted { get; private set; }
163
164     [MethodImpl(MethodImplOptions.AggressiveInlining)]
165     public Transaction(UInt64LinksTransactionsLayer layer)
166     {
167         _layer = layer;
168         if (_layer._currentTransactionId != 0)
169         {
170             throw new NotSupportedException("Nested transactions not supported.");
171         }
172         IsCommitted = false;
173         IsReverted = false;
174         _transitions = new Queue<Transition>();
175         SetCurrentTransaction(layer, this);
176     }
177
178     [MethodImpl(MethodImplOptions.AggressiveInlining)]
179     public void Commit()
180     {
181         EnsureTransactionAllowsWriteOperations(this);
182         while (_transitions.Count > 0)
183         {
184             var transition = _transitions.Dequeue();
185             _layer._transitions.Enqueue(transition);
186         }
187         _layer._lastCommittedTransactionId = _layer._currentTransactionId;
188         IsCommitted = true;
189     }
190
191     [MethodImpl(MethodImplOptions.AggressiveInlining)]
192     private void Revert()
193     {
194         EnsureTransactionAllowsWriteOperations(this);
195         var transitionsToRevert = new Transition[_transitions.Count];
196         _transitions.CopyTo(transitionsToRevert, 0);
197         for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
198         {
199             _layer.RevertTransition(transitionsToRevert[i]);
200         }
201         IsReverted = true;
202     }
203
204     [MethodImpl(MethodImplOptions.AggressiveInlining)]
205     public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
        ↪ Transaction transaction)
206     {
207         layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
208         layer._currentTransactionTransitions = transaction._transitions;
209         layer._currentTransaction = transaction;

```

```

210     }
211
212     [MethodImpl(MethodImplOptions.AggressiveInlining)]
213     public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
214     {
215         if (transaction.IsReverted)
216         {
217             throw new InvalidOperationException("Transation is reverted.");
218         }
219         if (transaction.IsCommitted)
220         {
221             throw new InvalidOperationException("Transation is committed.");
222         }
223     }
224
225     [MethodImpl(MethodImplOptions.AggressiveInlining)]
226     protected override void Dispose(bool manual, bool wasDisposed)
227     {
228         if (!wasDisposed && _layer != null && !_layer.Disposable.IsDisposed)
229         {
230             if (!IsCommitted && !IsReverted)
231             {
232                 Revert();
233             }
234             _layer.ResetCurrentTransation();
235         }
236     }
237 }
238
239 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
240
241 private readonly string _logAddress;
242 private readonly FileStream _log;
243 private readonly Queue<Transition> _transitions;
244 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
245 private Task _transitionsPusher;
246 private Transition _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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
288 public IList<ulong> GetLinkValue(ulong link) => _links.GetLink(link);
289
290 [MethodImpl(MethodImplOptions.AggressiveInlining)]
291 public override ulong Create(IList<ulong> restrictions)
292 {
293     var createdLinkIndex = _links.Create();
294     var createdLink = new Link<ulong>(_links.GetLink(createdLinkIndex));
295     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
296         ↪ default, createdLink));
297     return createdLinkIndex;
298 }
299
300 [MethodImpl(MethodImplOptions.AggressiveInlining)]
301 public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
302 {
303     var linkIndex = restrictions[_constants.IndexPart];
304     var beforeLink = new Link<ulong>(_links.GetLink(linkIndex));
305     linkIndex = _links.Update(restrictions, substitution);
306     var afterLink = new Link<ulong>(_links.GetLink(linkIndex));
307     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
308         ↪ beforeLink, afterLink));
309     return linkIndex;
310 }
311
312 [MethodImpl(MethodImplOptions.AggressiveInlining)]
313 public override void Delete(IList<ulong> restrictions)
314 {
315     var link = restrictions[_constants.IndexPart];
316     var deletedLink = new Link<ulong>(_links.GetLink(link));
317     _links.Delete(link);
318     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
319         ↪ deletedLink, default));
320 }
321
322 [MethodImpl(MethodImplOptions.AggressiveInlining)]
323 private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
324     ↪ _transitions;
325
326 [MethodImpl(MethodImplOptions.AggressiveInlining)]
327 private void CommitTransition(Transition transition)
328 {
329     if (_currentTransaction != null)
330     {
331         Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
332     }
333     var transitions = GetCurrentTransitions();
334     transitions.Enqueue(transition);
335 }
336
337 [MethodImpl(MethodImplOptions.AggressiveInlining)]
338 private void RevertTransition(Transition transition)
339 {
340     if (transition.After.IsNull()) // Revert Deletion with Creation
341     {
342         _links.Create();
343     }
344     else if (transition.Before.IsNull()) // Revert Creation with Deletion
345     {
346         _links.Delete(transition.After.Index);
347     }
348     else // Revert Update
349     {
350         _links.Update(new[] { transition.After.Index, transition.Before.Source,
351             ↪ transition.Before.Target });
352     }
353 }
354
355 [MethodImpl(MethodImplOptions.AggressiveInlining)]
356 private void ResetCurrentTransation()
357 {
358     _currentTransactionId = 0;
359     _currentTransactionTransitions = null;
360     _currentTransaction = null;
361 }
362
363 [MethodImpl(MethodImplOptions.AggressiveInlining)]
364 private void PushTransitions()

```

```

360 {
361     if (_log == null || _transitions == null)
362     {
363         return;
364     }
365     for (var i = 0; i < _transitions.Count; i++)
366     {
367         var transition = _transitions.Dequeue();
368
369         _log.Write(transition);
370         _lastCommittedTransition = transition;
371     }
372 }
373
374 [MethodImpl(MethodImplOptions.AggressiveInlining)]
375 private void TransitionsPusher()
376 {
377     while (!Disposable.IsDisposed && _transitionsPusher != null)
378     {
379         Thread.Sleep(DefaultPushDelay);
380         PushTransitions();
381     }
382 }
383
384 [MethodImpl(MethodImplOptions.AggressiveInlining)]
385 public Transaction BeginTransaction() => new Transaction(this);
386
387 [MethodImpl(MethodImplOptions.AggressiveInlining)]
388 private void DisposeTransitions()
389 {
390     try
391     {
392         var pusher = _transitionsPusher;
393         if (pusher != null)
394         {
395             _transitionsPusher = null;
396             pusher.Wait();
397         }
398         if (_transitions != null)
399         {
400             PushTransitions();
401         }
402         _log.DisposeIfPossible();
403         FileHelpers.WriteFirst(_logAddress, _lastCommittedTransition);
404     }
405     catch (Exception ex)
406     {
407         ex.Ignore();
408     }
409 }
410
411 #region DisposalBase
412
413 [MethodImpl(MethodImplOptions.AggressiveInlining)]
414 protected override void Dispose(bool manual, bool wasDisposed)
415 {
416     if (!wasDisposed)
417     {
418         DisposeTransitions();
419     }
420     base.Dispose(manual, wasDisposed);
421 }
422
423 #endregion
424 }
425 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Converters;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Unicode
7 {
8     public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
9     ↪ IConverter<char, TLink>
10     {
11         private static readonly UncheckedConverter<char, TLink> _charToAddressConverter =
12         ↪ UncheckedConverter<char, TLink>.Default;
13     }
14 }

```



```

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

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Converters;
4 using Platform.Data.Doublets.Sequences.Indexes;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Unicode
9 {
10     public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
11     ↪ IConverter<string, TLink>
12     {
13         private readonly IConverter<string, IList<TLink>> _stringToUnicodeSymbolListConverter;
14         private readonly IConverter<IList<TLink>, TLink> _unicodeSymbolListToSequenceConverter;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
18         ↪ IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
19         ↪ unicodeSymbolListToSequenceConverter) : base(links)
20         {
21             _stringToUnicodeSymbolListConverter = stringToUnicodeSymbolListConverter;
22             _unicodeSymbolListToSequenceConverter = unicodeSymbolListToSequenceConverter;
23         }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
27         ↪ IList<TLink>> stringToUnicodeSymbolListConverter, ISequenceIndex<TLink> index,
28         ↪ IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
29         ↪ unicodeSequenceMarker)
30         : this(links, stringToUnicodeSymbolListConverter, new
31         ↪ UnicodeSymbolsListToUnicodeSequenceConverter<TLink>(links, index,
32         ↪ listToSequenceLinkConverter, unicodeSequenceMarker)) { }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
36         ↪ charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
37         ↪ TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker)
38         : this(links, new
39         ↪ StringToUnicodeSymbolsListConverter<TLink>(charToUnicodeSymbolConverter), index,
40         ↪ listToSequenceLinkConverter, unicodeSequenceMarker) { }
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
44         ↪ charToUnicodeSymbolConverter, IConverter<IList<TLink>, TLink>
45         ↪ listToSequenceLinkConverter, TLink unicodeSequenceMarker)
46         : this(links, charToUnicodeSymbolConverter, new Unindex<TLink>(),
47         ↪ listToSequenceLinkConverter, unicodeSequenceMarker) { }
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
51         ↪ IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
52         ↪ listToSequenceLinkConverter, TLink unicodeSequenceMarker)
53         : this(links, stringToUnicodeSymbolListConverter, new Unindex<TLink>(),
54         ↪ listToSequenceLinkConverter, unicodeSequenceMarker) { }
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

39     public TLink Convert(string source)
40     {
41         var elements = _stringToUnicodeSymbolListConverter.Convert(source);
42         return _unicodeSymbolListToSequenceConverter.Convert(elements);
43     }
44 }
45 }

```

1.170 ./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSymbolsListConverter.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Converters;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Unicode
8  {
9      public class StringToUnicodeSymbolsListConverter<TLink> : IConverter<string, IList<TLink>>
10     {
11         private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public StringToUnicodeSymbolsListConverter(IConverter<char, TLink>
15             ↪ charToUnicodeSymbolConverter) => _charToUnicodeSymbolConverter =
16             ↪ charToUnicodeSymbolConverter;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public IList<TLink> Convert(string source)
20         {
21             var elements = new TLink[source.Length];
22             for (var i = 0; i < elements.Length; i++)
23             {
24                 elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
25             }
26             return elements;
27         }
28     }
29 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Globalization;
4  using System.Runtime.CompilerServices;
5  using System.Text;
6  using Platform.Data.Sequences;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Unicode
11 {
12     public class UnicodeMap
13     {
14         public static readonly ulong FirstCharLink = 1;
15         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
16         public static readonly ulong MapSize = 1 + char.MaxValue;
17
18         private readonly ILinks<ulong> _links;
19         private bool _initialized;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public UnicodeMap(ILinks<ulong> links) => _links = links;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public static UnicodeMap InitNew(ILinks<ulong> links)
26         {
27             var map = new UnicodeMap(links);
28             map.Init();
29             return map;
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public void Init()
34         {
35             if (_initialized)
36             {
37                 return;
38             }
39             _initialized = true;
40             var firstLink = _links.CreatePoint();

```

```

41     if (firstLink != FirstCharLink)
42     {
43         _links.Delete(firstLink);
44     }
45     else
46     {
47         for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
48         {
49             // From NIL to It (NIL -> Character) transformation meaning, (or infinite
50             // ↪ amount of NIL characters before actual Character)
51             var createdLink = _links.CreatePoint();
52             _links.Update(createdLink, firstLink, createdLink);
53             if (createdLink != i)
54             {
55                 throw new InvalidOperationException("Unable to initialize UTF 16
56                 ↪ table.");
57             }
58         }
59     }
60     // 0 - null link
61     // 1 - nil character (0 character)
62     // ...
63     // 65536 (0(1) + 65535 = 65536 possible values)
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     public static ulong FromCharToLink(char character) => (ulong)character + 1;
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     public static char FromLinkToChar(ulong link) => (char)(link - 1);
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     public static bool IsCharLink(ulong link) => link <= MapSize;
73
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     public static string FromLinksToString(IList<ulong> linksList)
76     {
77         var sb = new StringBuilder();
78         for (int i = 0; i < linksList.Count; i++)
79         {
80             sb.Append(FromLinkToChar(linksList[i]));
81         }
82         return sb.ToString();
83     }
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
87     {
88         var sb = new StringBuilder();
89         if (links.Exists(link))
90         {
91             StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
92             x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
93             ↪ element =>
94             {
95                 sb.Append(FromLinkToChar(element));
96                 return true;
97             }
98         }
99         return sb.ToString();
100     }
101
102     [MethodImpl(MethodImplOptions.AggressiveInlining)]
103     public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
104     ↪ chars.Length);
105
106     [MethodImpl(MethodImplOptions.AggressiveInlining)]
107     public static ulong[] FromCharsToLinkArray(char[] chars, int count)
108     {
109         // char array to ulong array
110         var linksSequence = new ulong[count];
111         for (var i = 0; i < count; i++)
112         {
113             linksSequence[i] = FromCharToLink(chars[i]);
114         }
115         return linksSequence;
116     }

```

```

115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 public static ulong[] FromStringToLinkArray(string sequence)
117 {
118     // char array to ulong array
119     var linksSequence = new ulong[sequence.Length];
120     for (var i = 0; i < sequence.Length; i++)
121     {
122         linksSequence[i] = FromCharToLink(sequence[i]);
123     }
124     return linksSequence;
125 }
126
127 [MethodImpl(MethodImplOptions.AggressiveInlining)]
128 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
129 {
130     var result = new List<ulong[]>();
131     var offset = 0;
132     while (offset < sequence.Length)
133     {
134         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
135         var relativeLength = 1;
136         var absoluteLength = offset + relativeLength;
137         while (absoluteLength < sequence.Length &&
138             currentCategory ==
139             CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
140         {
141             relativeLength++;
142             absoluteLength++;
143         }
144         // char array to ulong array
145         var innerSequence = new ulong[relativeLength];
146         var maxLength = offset + relativeLength;
147         for (var i = offset; i < maxLength; i++)
148         {
149             innerSequence[i - offset] = FromCharToLink(sequence[i]);
150         }
151         result.Add(innerSequence);
152         offset += relativeLength;
153     }
154     return result;
155 }
156
157 [MethodImpl(MethodImplOptions.AggressiveInlining)]
158 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
159 {
160     var result = new List<ulong[]>();
161     var offset = 0;
162     while (offset < array.Length)
163     {
164         var relativeLength = 1;
165         if (array[offset] <= LastCharLink)
166         {
167             var currentCategory =
168             CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
169             var absoluteLength = offset + relativeLength;
170             while (absoluteLength < array.Length &&
171                 array[absoluteLength] <= LastCharLink &&
172                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
173                 CharUnicodeInfo.GetUnicodeCategory(array[absoluteLength])))
174             {
175                 relativeLength++;
176                 absoluteLength++;
177             }
178         }
179         else
180         {
181             var absoluteLength = offset + relativeLength;
182             while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
183             {
184                 relativeLength++;
185                 absoluteLength++;
186             }
187         }
188         // copy array
189         var innerSequence = new ulong[relativeLength];
190         var maxLength = offset + relativeLength;
191         for (var i = offset; i < maxLength; i++)
192         {

```

```

191         innerSequence[i - offset] = array[i];
192     }
193     result.Add(innerSequence);
194     offset += relativeLength;
195 }
196 return result;
197 }
198 }
199 }

```

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

```

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

```

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

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4  using Platform.Converters;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Unicode
9  {
10     public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
11         ↪ IConverter<TLink, char>
12     {
13         private static readonly UncheckedConverter<TLink, char> _addressToCharConverter =
14             ↪ UncheckedConverter<TLink, char>.Default;
15
16         private readonly IConverter<TLink> _numberToAddressConverter;
17         private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
21             ↪ numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
22             ↪ base(links)
23         {
24         }
25     }
26 }

```

```

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

```

1.174 ./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Converters;
4  using Platform.Data.Doublets.Sequences.Indexes;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Unicode
9  {
10     public class UnicodeSymbolsListToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
↳ IConverter<IList<TLink>, TLink>
11     {
12         private readonly ISequenceIndex<TLink> _index;
13         private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
14         private readonly TLink _unicodeSequenceMarker;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
↳ ISequenceIndex<TLink> index, IConverter<IList<TLink>, TLink>
↳ listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
18         {
19             _index = index;
20             _listToSequenceLinkConverter = listToSequenceLinkConverter;
21             _unicodeSequenceMarker = unicodeSequenceMarker;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
↳ IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
↳ unicodeSequenceMarker)
26             : this(links, new Unindex<TLink>(), listToSequenceLinkConverter,
↳ unicodeSequenceMarker) { }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public TLink Convert(IList<TLink> list)
30         {
31             _index.Add(list);
32             var sequence = _listToSequenceLinkConverter.Convert(list);
33             return _links.GetOrCreate(sequence, _unicodeSequenceMarker);
34         }
35     }
36 }

```

1.175 ./csharp/Platform.Data.Doublets.Tests/BigIntegerConvertersTests.cs

```

1  using System.Collections.Generic;
2  using System.Numerics;
3  using Platform.Data.Doublets.Memory;
4  using Platform.Data.Doublets.Memory.United.Generic;
5  using Platform.Data.Doublets.Numbers.Raw;
6  using Platform.Data.Doublets.Sequences.Converters;
7  using Platform.Data.Numbers.Raw;
8  using Platform.Memory;
9  using Xunit;
10 using TLink = System.UInt64;
11
12 namespace Platform.Data.Doublets.Tests
13 {
14     public class BigIntegerConvertersTests
15     {
16         public ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new IO.TemporaryFile());

```

```

17
18 public ILinks<TLink> CreateLinks<TLink>(string dataDbFilename)
19 {
20     var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
21         ↪ true);
22     return new UnitedMemoryLinks<TLink>(new
23         ↪ FileMappedResizableDirectMemory(dataDbFilename),
24         ↪ UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
25         ↪ IndexTreeType.Default);
26 }
27
28 [Fact]
29 public void DecimalMaxValueTest()
30 {
31     var links = CreateLinks();
32     BigInteger bigInteger = new(decimal.MaxValue);
33     TLink negativeNumberMarker = links.Create();
34     AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
35     RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
36     BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
37     BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
38         ↪ = new(links, addressToRawNumberConverter, listToSequenceConverter,
39         ↪ negativeNumberMarker);
40     RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
41         ↪ = new(links, numberToAddressConverter, negativeNumberMarker);
42     var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
43     var bigIntFromSequence =
44         ↪ rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
45     Assert.Equal(bigInteger, bigIntFromSequence);
46 }
47
48 [Fact]
49 public void DecimalMinValueTest()
50 {
51     var links = CreateLinks();
52     BigInteger bigInteger = new(decimal.MinValue);
53     TLink negativeNumberMarker = links.Create();
54     AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
55     RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
56     BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
57     BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
58         ↪ = new(links, addressToRawNumberConverter, listToSequenceConverter,
59         ↪ negativeNumberMarker);
60     RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
61         ↪ = new(links, numberToAddressConverter, negativeNumberMarker);
62     var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
63     var bigIntFromSequence =
64         ↪ rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
65     Assert.Equal(bigInteger, bigIntFromSequence);
66 }
67
68 [Fact]
69 public void ZeroValueTest()
70 {
71     var links = CreateLinks();
72     BigInteger bigInteger = new(0);
73     TLink negativeNumberMarker = links.Create();
74     AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
75     RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
76     BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
77     BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
78         ↪ = new(links, addressToRawNumberConverter, listToSequenceConverter,
79         ↪ negativeNumberMarker);
80     RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
81         ↪ = new(links, numberToAddressConverter, negativeNumberMarker);
82     var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
83     var bigIntFromSequence =
84         ↪ rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
85     Assert.Equal(bigInteger, bigIntFromSequence);
86 }
87
88 [Fact]
89 public void OneValueTest()
90 {
91     var links = CreateLinks();
92     BigInteger bigInteger = new(1);
93     TLink negativeNumberMarker = links.Create();

```

```

78     AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
79     RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
80     BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
81     BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
    ↪ = new(links, addressToRawNumberConverter, listToSequenceConverter,
    ↪ negativeNumberMarker);
82     RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
    ↪ = new(links, numberToAddressConverter, negativeNumberMarker);
83     var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
84     var bigIntFromSequence =
    ↪ rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
85     Assert.Equal(bigInteger, bigIntFromSequence);
86 }
87 }
88 }

```

1.176 ./csharp/Platform.Data.Doublets.Tests/DefaultSequenceAppenderTests.cs

```

1  using System.Collections.Generic;
2  using Platform.Collections.Stacks;
3  using Platform.Data.Doublets.Memory;
4  using Platform.Data.Doublets.Memory.United.Generic;
5  using Platform.Data.Doublets.Sequences;
6  using Platform.Data.Doublets.Sequences.HeightProviders;
7  using Platform.Data.Numbers.Raw;
8  using Platform.Interfaces;
9  using Platform.Memory;
10 using Platform.Numbers;
11 using Xunit;
12 using Xunit.Abstractions;
13 using TLink = System.UInt64;
14
15 namespace Platform.Data.Doublets.Tests
16 {
17     public class DefaultSequenceAppenderTests
18     {
19         private readonly ITestOutputHelper _output;
20
21         public DefaultSequenceAppenderTests(ITestOutputHelper output)
22         {
23             _output = output;
24         }
25         public static ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new IO.TemporaryFile());
26
27         public static ILinks<TLink> CreateLinks<TLink>(string dataDBFilename)
28         {
29             var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
    ↪ true);
30             return new UnitedMemoryLinks<TLink>(new
    ↪ FileMappedResizableDirectMemory(dataDBFilename),
    ↪ UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
    ↪ IndexTreeType.Default);
31         }
32
33         public class ValueCriterionMatcher<TLink> : ICriterionMatcher<TLink>
34         {
35             public readonly ILinks<TLink> Links;
36             public readonly TLink Marker;
37             public ValueCriterionMatcher(ILinks<TLink> links, TLink marker)
38             {
39                 Links = links;
40                 Marker = marker;
41             }
42
43             public bool IsMatched(TLink link) =>
    ↪ EqualityComparer<TLink>.Default.Equals(Links.GetSource(link), Marker);
44         }
45
46         [Fact]
47         public void AppendArrayBug()
48         {
49             ILinks<TLink> links = CreateLinks();
50             TLink zero = default;
51             var markerIndex = Arithmetic.Increment(zero);
52             var meaningRoot = links.GetOrCreate(markerIndex, markerIndex);
53             var sequence = links.Create();
54             sequence = links.Update(sequence, meaningRoot, sequence);
55             var appendant = links.Create();
56             appendant = links.Update(appendant, meaningRoot, appendant);
57             ValueCriterionMatcher<TLink> valueCriterionMatcher = new(links, meaningRoot);

```



```

58     DefaultSequenceRightHeightProvider<ulong> defaultSequenceRightHeightProvider =
        ↳ new(links, valueCriterionMatcher);
59     DefaultSequenceAppender<TLink> defaultSequenceAppender = new(links, new
        ↳ DefaultStack<ulong>(), defaultSequenceRightHeightProvider);
60     var newArray = defaultSequenceAppender.Append(sequence, appendant);
61     var output = links.FormatStructure(newArray, link => link.IsFullPoint(), true);
62     Assert.Equal("(4:(2:1 2) (3:1 3))", output);
63 }
64 }
65 }

```

1.177 ./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
        ↳ MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
        ↳ implementation of tree cuts out 5 bits from the address space.
34             Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
        ↳ stMultipleRandomCreationsAndDeletions(100));
35             Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
        ↳ MultipleRandomCreationsAndDeletions(100));
36             Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
        ↳ tMultipleRandomCreationsAndDeletions(100));
37         }
38
39         private static void Using<TLink>(Action<ILinks<TLink>> action)
40         {
41             using (var scope = new Scope<Types<HeapResizableDirectMemory,
        ↳ UnitedMemoryLinks<TLink>>>())
42             {
43                 action(scope.Use<ILinks<TLink>>());
44             }
45         }
46     }
47 }

```

1.178 ./csharp/Platform.Data.Doublets.Tests/ILinksExtensionsTests.cs

```

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

```

```

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

```

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

```

1  using Xunit;
2
3  namespace Platform.Data.Doublets.Tests
4  {
5      public static class LinksConstantsTests
6      {
7          [Fact]
8          public static void ExternalReferencesTest()
9          {
10              LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
11                  ↳ (long.MaxValue + 1UL, ulong.MaxValue));
12
13              //var minimum = new Hybrid<ulong>(0, isExternal: true);
14              var minimum = new Hybrid<ulong>(1, isExternal: true);
15              var maximum = new Hybrid<ulong>(long.MaxValue, isExternal: true);
16
17              Assert.True(constants.IsExternalReference(minimum));
18              Assert.True(constants.IsExternalReference(maximum));
19          }
20 }

```

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

```

1  using System;
2  using System.Linq;
3  using Xunit;
4  using Platform.Collections.Stacks;
5  using Platform.Collections.Arrays;
6  using Platform.Memory;
7  using Platform.Data.Numbers.Raw;
8  using Platform.Data.Doublets.Sequences;
9  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
10 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
11 using Platform.Data.Doublets.Sequences.Converters;
12 using Platform.Data.Doublets.PropertyOperators;
13 using Platform.Data.Doublets.Incrementers;
14 using Platform.Data.Doublets.Sequences.Walkers;
15 using Platform.Data.Doublets.Sequences.Indexes;
16 using Platform.Data.Doublets.Unicode;
17 using Platform.Data.Doublets.Numbers.Unary;
18 using Platform.Data.Doublets.Decorators;
19 using Platform.Data.Doublets.Memory.United.Specific;
20 using Platform.Data.Doublets.Memory;
21
22 namespace Platform.Data.Doublets.Tests
23 {
24     public static class OptimalVariantSequenceTests
25     {
26         private static readonly string _sequenceExample = "зеленела зелёная зелень";
27         private static readonly string _loremIpsumExample = @"Lorem ipsum dolor sit amet,
28             ↳ consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore
29             ↳ magna aliqua.
30             Facilisi nullam vehicula ipsum a arcu cursus vitae congue mauris.
31             Et malesuada fames ac turpis egestas sed.
32             Eget velit aliquet sagittis id consectetur purus.
33             Dignissim cras tincidunt lobortis feugiat vivamus.
34             Vitae aliquet nec ullamcorper sit.
35             Lectus quam id leo in vitae.
36             Tortor dignissim convallis aenean et tortor at risus viverra adipiscing.
37             Sed risus ultricies tristique nulla aliquet enim tortor at auctor.
38             Integer eget aliquet nibh praesent tristique.
39             Vitae congue eu consequat ac felis donec et odio.
40             Tristique et egestas quis ipsum suspendisse.
41             Suspendisse potenti nullam ac tortor vitae purus faucibus ornare.
42             Nulla facilisi etiam dignissim diam quis enim lobortis scelerisque.
43             Imperdiet proin fermentum leo vel orci.
44             In ante metus dictum at tempor commodo.
45             Nisi lacus sed viverra tellus in.
46             Quam vulputate dignissim suspendisse in.
47             Elit scelerisque mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus.
48             Gravida cum sociis natoque penatibus et magnis dis parturient.
49             Risus quis varius quam quisque id diam.

```

```

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

```

```

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

```

```

171     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
172         ↪ sequenceToItsLocalElementLevelsConverter);
173
174     var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
175         ↪ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
176
177     var unicodeSequencesOptions = new SequencesOptions<ulong>()
178     {
179         UseSequenceMarker = true,
180         SequenceMarkerLink = unicodeSequenceMarker,
181         UseIndex = true,
182         Index = index,
183         LinksToSequenceConverter = optimalVariantConverter,
184         Walker = walker,
185         UseGarbageCollection = true
186     };
187
188     var unicodeSequences = new Sequences.Sequences(new
189         ↪ SynchronizedLinks<ulong>(links), unicodeSequencesOptions);
190
191     // Create some sequences
192     var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
193         ↪ StringSplitOptions.RemoveEmptyEntries);
194     var arrays = strings.Select(x => x.Select(y =>
195         ↪ addressToNumberConverter.Convert(y)).ToArray()).ToArray();
196     for (int i = 0; i < arrays.Length; i++)
197     {
198         unicodeSequences.Create(arrays[i].ShiftRight());
199     }
200
201     var linksCountAfterCreation = links.Count();
202
203     // get list of sequences links
204     // for each sequence link
205     //     create new sequence version
206     //     if new sequence is not the same as sequence link
207     //         delete sequence link
208     //         collect garbadge
209     unicodeSequences.CompactAll();
210
211     var linksCountAfterCompactification = links.Count();
212
213     Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
214 }
215 }
216 }

```

1.181 ./csharp/Platform.Data.Doublets.Tests/RationalNumbersTests.cs

```

1  using Platform.Data.Doublets.Memory;
2  using Platform.Data.Doublets.Memory.United.Generic;
3  using Platform.Data.Doublets.Numbers.Rational;
4  using Platform.Data.Doublets.Numbers.Raw;
5  using Platform.Data.Doublets.Sequences.Converters;
6  using Platform.Data.Numbers.Raw;
7  using Platform.Memory;
8  using Xunit;
9  using TLink = System.UInt64;
10
11 namespace Platform.Data.Doublets.Tests
12 {
13     public class RationalNumbersTests
14     {
15         public ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new IO.TemporaryFile());
16
17         public ILinks<TLink> CreateLinks<TLink>(string dataDbFilename)
18         {
19             var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
20                 ↪ true);
21             return new UnitedMemoryLinks<TLink>(new
22                 ↪ FileMappedResizableDirectMemory(dataDbFilename),
23                 ↪ UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
24                 ↪ IndexTreeType.Default);
25         }
26
27         [Fact]
28         public void DecimalMinValueTest()
29         {
30             const decimal @decimal = decimal.MinValue;
31             var links = CreateLinks();

```

```

28     TLink negativeNumberMarker = links.Create();
29     AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
30     RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
31     BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
32     BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
    ↪ = new(links, addressToRawNumberConverter, balancedVariantConverter,
    ↪ negativeNumberMarker);
33     RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
    ↪ = new(links, numberToAddressConverter, negativeNumberMarker);
34     DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
    ↪ bigIntegerToRawNumberSequenceConverter);
35     RationalToDecimalConverter<TLink> rationalToDecimalConverter = new(links,
    ↪ rawNumberSequenceToBigIntegerConverter);
36     var rationalNumber = decimalToRationalConverter.Convert(@decimal);
37     var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
38     Assert.Equal(@decimal, decimalFromRational);
39 }
40
41 [Fact]
42 public void DecimalMaxValueTest()
43 {
44     const decimal @decimal = decimal.MaxValue;
45     var links = CreateLinks();
46     TLink negativeNumberMarker = links.Create();
47     AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
48     RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
49     BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
50     BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
    ↪ = new(links, addressToRawNumberConverter, balancedVariantConverter,
    ↪ negativeNumberMarker);
51     RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
    ↪ = new(links, numberToAddressConverter, negativeNumberMarker);
52     DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
    ↪ bigIntegerToRawNumberSequenceConverter);
53     RationalToDecimalConverter<TLink> rationalToDecimalConverter = new(links,
    ↪ rawNumberSequenceToBigIntegerConverter);
54     var rationalNumber = decimalToRationalConverter.Convert(@decimal);
55     var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
56     Assert.Equal(@decimal, decimalFromRational);
57 }
58
59 [Fact]
60 public void DecimalPositiveHalfTest()
61 {
62     const decimal @decimal = 0.5M;
63     var links = CreateLinks();
64     TLink negativeNumberMarker = links.Create();
65     AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
66     RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
67     BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
68     BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
    ↪ = new(links, addressToRawNumberConverter, balancedVariantConverter,
    ↪ negativeNumberMarker);
69     RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
    ↪ = new(links, numberToAddressConverter, negativeNumberMarker);
70     DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
    ↪ bigIntegerToRawNumberSequenceConverter);
71     RationalToDecimalConverter<TLink> rationalToDecimalConverter = new(links,
    ↪ rawNumberSequenceToBigIntegerConverter);
72     var rationalNumber = decimalToRationalConverter.Convert(@decimal);
73     var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
74     Assert.Equal(@decimal, decimalFromRational);
75 }
76
77 [Fact]
78 public void DecimalNegativeHalfTest()
79 {
80     const decimal @decimal = -0.5M;
81     var links = CreateLinks();
82     TLink negativeNumberMarker = links.Create();
83     AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
84     RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
85     BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
86     BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
    ↪ = new(links, addressToRawNumberConverter, balancedVariantConverter,
    ↪ negativeNumberMarker);
87     RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
    ↪ = new(links, numberToAddressConverter, negativeNumberMarker);

```

```

88     DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
89         ↪ BigIntegerToRawNumberSequenceConverter);
90     RationalToDecimalConverter<TLink> rationalToDecimalConverter = new(links,
91         ↪ rawNumberSequenceToBigIntegerConverter);
92     var rationalNumber = decimalToRationalConverter.Convert(@decimal);
93     var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
94     Assert.Equal(@decimal, decimalFromRational);
95 }
96
97 [Fact]
98 public void DecimalOneTest()
99 {
100     const decimal @decimal = 1;
101     var links = CreateLinks();
102     TLink negativeNumberMarker = links.Create();
103     AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
104     RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
105     BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
106     BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
107         ↪ = new(links, addressToRawNumberConverter, balancedVariantConverter,
108         ↪ negativeNumberMarker);
109     RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
110         ↪ = new(links, numberToAddressConverter, negativeNumberMarker);
111     DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
112         ↪ BigIntegerToRawNumberSequenceConverter);
113     RationalToDecimalConverter<TLink> rationalToDecimalConverter = new(links,
114         ↪ rawNumberSequenceToBigIntegerConverter);
115     var rationalNumber = decimalToRationalConverter.Convert(@decimal);
116     var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
117     Assert.Equal(@decimal, decimalFromRational);
118 }
119
120 [Fact]
121 public void DecimalMinusOneTest()
122 {
123     const decimal @decimal = -1;
124     var links = CreateLinks();
125     TLink negativeNumberMarker = links.Create();
126     AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
127     RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
128     BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
129     BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
130         ↪ = new(links, addressToRawNumberConverter, balancedVariantConverter,
131         ↪ negativeNumberMarker);
132     RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
133         ↪ = new(links, numberToAddressConverter, negativeNumberMarker);
134     DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
135         ↪ BigIntegerToRawNumberSequenceConverter);
136     RationalToDecimalConverter<TLink> rationalToDecimalConverter = new(links,
137         ↪ rawNumberSequenceToBigIntegerConverter);
138     var rationalNumber = decimalToRationalConverter.Convert(@decimal);
139     var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
140     Assert.Equal(@decimal, decimalFromRational);
141 }
142 }
143 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Diagnostics;
4 using System.Linq;
5 using Xunit;
6 using Platform.Data.Sequences;
7 using Platform.Data.Doublets.Sequences.Converters;
8 using Platform.Data.Doublets.Sequences.Walkers;
9 using Platform.Data.Doublets.Sequences;
10
11 namespace Platform.Data.Doublets.Tests
12 {
13     public static class ReadSequenceTests
14     {
15         [Fact]
16         public static void ReadSequenceTest()
17         {
18             const long sequenceLength = 2000;
19
20             using (var scope = new TempLinksTestScope(useSequences: false))
21             {

```

```

22     var links = scope.Links;
23     var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong> {
24         ↳ Walker = new LeveledSequenceWalker<ulong>(links) });
25
26     var sequence = new ulong[sequenceLength];
27     for (var i = 0; i < sequenceLength; i++)
28     {
29         sequence[i] = links.Create();
30     }
31
32     var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
33
34     var sw1 = Stopwatch.StartNew();
35     var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
36
37     var sw2 = Stopwatch.StartNew();
38     var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();
39
40     var sw3 = Stopwatch.StartNew();
41     var readSequence2 = new List<ulong>();
42     SequenceWalker.WalkRight(balancedVariant,
43                             links.GetSource,
44                             links.GetTarget,
45                             links.IsPartialPoint,
46                             readSequence2.Add);
47     sw3.Stop();
48
49     Assert.True(sequence.SequenceEqual(readSequence1));
50
51     Assert.True(sequence.SequenceEqual(readSequence2));
52
53     // Assert.True(sw2.Elapsed < sw3.Elapsed);
54
55     Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
56     ↳ {sw2.Elapsed}");
57
58     for (var i = 0; i < sequenceLength; i++)
59     {
60         links.Delete(sequence[i]);
61     }
62 }
63 }

```

1.183 ./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         }
36     }
37 }

```



```

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

```

1.184 ./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 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Linq;
5  using Xunit;
6  using Platform.Collections;
7  using Platform.Collections.Arrays;
8  using Platform.Random;
9  using Platform.IO;
10 using Platform.Singletons;
11 using Platform.Data.Doublets.Sequences;
12 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
13 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
14 using Platform.Data.Doublets.Sequences.Converters;
15 using Platform.Data.Doublets.Unicode;
16
17 namespace Platform.Data.Doublets.Tests
18 {
19     public static class SequencesTests
20     {
21         private static readonly LinksConstants<ulong> _constants =
22             ↪ Default<LinksConstants<ulong>>.Instance;
23
24         static SequencesTests()
25         {
26             // Trigger static constructor to not mess with performance measurements
27             _ = BitString.GetBitMaskFromIndex(1);
28         }
29
30         [Fact]
31         public static void CreateAllVariantsTest()
32         {
33             const long sequenceLength = 8;
34
35             using (var scope = new TempLinksTestScope(useSequences: true))
36             {
37                 var links = scope.Links;
38                 var sequences = scope.Sequences;
39
40                 var sequence = new ulong[sequenceLength];
41                 for (var i = 0; i < sequenceLength; i++)
42                 {
43                     sequence[i] = links.Create();
44                 }
45
46                 var sw1 = Stopwatch.StartNew();
47                 var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
48
49                 var sw2 = Stopwatch.StartNew();
50                 var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
51
52                 Assert.True(results1.Count > results2.Length);
53                 Assert.True(sw1.Elapsed > sw2.Elapsed);
54
55                 for (var i = 0; i < sequenceLength; i++)
56                 {
57                     links.Delete(sequence[i]);
58                 }
59
60                 Assert.True(links.Count() == 0);
61             }
62         }
63     }
64 }

```

```

61     }
62
63     //[Fact]
64     //public void CUDTest()
65     //{
66     //    var tempFilename = Path.GetTempFileName();
67
68     //    const long sequenceLength = 8;
69
70     //    const ulong itself = LinksConstants.Itself;
71
72     //    using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
73     //        ↪ DefaultLinksSizeStep))
74     //    using (var links = new Links(memoryAdapter))
75     //    {
76     //        var sequence = new ulong[sequenceLength];
77     //        for (var i = 0; i < sequenceLength; i++)
78     //            sequence[i] = links.Create(itself, itself);
79
80     //        SequencesOptions o = new SequencesOptions();
81
82     //        TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
83     //        o.
84
85     //        var sequences = new Sequences(links);
86
87     //        var sw1 = Stopwatch.StartNew();
88     //        var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
89
90     //        var sw2 = Stopwatch.StartNew();
91     //        var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
92
93     //        Assert.True(results1.Count > results2.Length);
94     //        Assert.True(sw1.Elapsed > sw2.Elapsed);
95
96     //        for (var i = 0; i < sequenceLength; i++)
97     //            links.Delete(sequence[i]);
98     //    }
99
100    //    File.Delete(tempFilename);
101    //}
102
103    [Fact]
104    public static void AllVariantsSearchTest()
105    {
106        const long sequenceLength = 8;
107
108        using (var scope = new TempLinksTestScope(useSequences: true))
109        {
110            var links = scope.Links;
111            var sequences = scope.Sequences;
112
113            var sequence = new ulong[sequenceLength];
114            for (var i = 0; i < sequenceLength; i++)
115            {
116                sequence[i] = links.Create();
117            }
118
119            var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
120
121            //for (int i = 0; i < createResults.Length; i++)
122            //    sequences.Create(createResults[i]);
123
124            var sw0 = Stopwatch.StartNew();
125            var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
126
127            var sw1 = Stopwatch.StartNew();
128            var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
129
130            var sw2 = Stopwatch.StartNew();
131            var searchResults2 = sequences.Each1(sequence); sw2.Stop();
132
133            var sw3 = Stopwatch.StartNew();
134            var searchResults3 = sequences.Each(sequence.ShiftRight()); sw3.Stop();
135
136            var intersection0 = createResults.Intersect(searchResults0).ToList();
137            Assert.True(intersection0.Count == searchResults0.Count);
138            Assert.True(intersection0.Count == createResults.Length);
139
140            var intersection1 = createResults.Intersect(searchResults1).ToList();

```

```

140     Assert.True(intersection1.Count == searchResults1.Count);
141     Assert.True(intersection1.Count == createResults.Length);
142
143     var intersection2 = createResults.Intersect(searchResults2).ToList();
144     Assert.True(intersection2.Count == searchResults2.Count);
145     Assert.True(intersection2.Count == createResults.Length);
146
147     var intersection3 = createResults.Intersect(searchResults3).ToList();
148     Assert.True(intersection3.Count == searchResults3.Count);
149     Assert.True(intersection3.Count == createResults.Length);
150
151     for (var i = 0; i < sequenceLength; i++)
152     {
153         links.Delete(sequence[i]);
154     }
155 }
156 }
157
158 [Fact]
159 public static void BalancedVariantSearchTest()
160 {
161     const long sequenceLength = 200;
162
163     using (var scope = new TempLinksTestScope(useSequences: true))
164     {
165         var links = scope.Links;
166         var sequences = scope.Sequences;
167
168         var sequence = new ulong[sequenceLength];
169         for (var i = 0; i < sequenceLength; i++)
170         {
171             sequence[i] = links.Create();
172         }
173
174         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
175
176         var sw1 = Stopwatch.StartNew();
177         var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
178
179         var sw2 = Stopwatch.StartNew();
180         var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
181
182         var sw3 = Stopwatch.StartNew();
183         var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
184
185         // На количестве в 200 элементов это будет занимать вечность
186         //var sw4 = Stopwatch.StartNew();
187         //var searchResults4 = sequences.Each(sequence); sw4.Stop();
188
189         Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
190
191         Assert.True(searchResults3.Count == 1 && balancedVariant ==
192             ↪ searchResults3.First());
193
194         //Assert.True(sw1.Elapsed < sw2.Elapsed);
195
196         for (var i = 0; i < sequenceLength; i++)
197         {
198             links.Delete(sequence[i]);
199         }
200     }
201 }
202
203 [Fact]
204 public static void AllPartialVariantsSearchTest()
205 {
206     const long sequenceLength = 8;
207
208     using (var scope = new TempLinksTestScope(useSequences: true))
209     {
210         var links = scope.Links;
211         var sequences = scope.Sequences;
212
213         var sequence = new ulong[sequenceLength];
214         for (var i = 0; i < sequenceLength; i++)
215         {
216             sequence[i] = links.Create();
217         }
218
219         var createResults = sequences.CreateAllVariants2(sequence);

```

```

219
220 //var createResultsStrings = createResults.Select(x => x + ": " +
    ↳ sequences.FormatSequence(x)).ToList();
221 //Global.Trash = createResultsStrings;
222
223 var partialSequence = new ulong[sequenceLength - 2];
224
225 Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
226
227 var sw1 = Stopwatch.StartNew();
228 var searchResults1 =
    ↳ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
229
230 var sw2 = Stopwatch.StartNew();
231 var searchResults2 =
    ↳ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
232
233 //var sw3 = Stopwatch.StartNew();
234 //var searchResults3 =
    ↳ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
235
236 var sw4 = Stopwatch.StartNew();
237 var searchResults4 =
    ↳ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
238
239 //Global.Trash = searchResults3;
240
241 //var searchResults1Strings = searchResults1.Select(x => x + ": " +
    ↳ sequences.FormatSequence(x)).ToList();
242 //Global.Trash = searchResults1Strings;
243
244 var intersection1 = createResults.Intersect(searchResults1).ToList();
245 Assert.True(intersection1.Count == createResults.Length);
246
247 var intersection2 = createResults.Intersect(searchResults2).ToList();
248 Assert.True(intersection2.Count == createResults.Length);
249
250 var intersection4 = createResults.Intersect(searchResults4).ToList();
251 Assert.True(intersection4.Count == createResults.Length);
252
253 for (var i = 0; i < sequenceLength; i++)
254 {
255     links.Delete(sequence[i]);
256 }
257 }
258
259 [Fact]
260 public static void BalancedPartialVariantsSearchTest()
261 {
262     const long sequenceLength = 200;
263
264     using (var scope = new TempLinksTestScope(useSequences: true))
265     {
266         var links = scope.Links;
267         var sequences = scope.Sequences;
268
269         var sequence = new ulong[sequenceLength];
270         for (var i = 0; i < sequenceLength; i++)
271         {
272             sequence[i] = links.Create();
273         }
274
275         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
276         var balancedVariant = balancedVariantConverter.Convert(sequence);
277
278         var partialSequence = new ulong[sequenceLength - 2];
279
280         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
281
282         var sw1 = Stopwatch.StartNew();
283         var searchResults1 =
284             ↳ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
285
286         var sw2 = Stopwatch.StartNew();
287         var searchResults2 =
288             ↳ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
289

```

```

290         Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
291
292         Assert.True(searchResults2.Count == 1 && balancedVariant ==
293             ↳ searchResults2.First());
294
295         for (var i = 0; i < sequenceLength; i++)
296         {
297             links.Delete(sequence[i]);
298         }
299     }
300
301     [Fact(Skip = "Correct implementation is pending")]
302     public static void PatternMatchTest()
303     {
304         var zeroOrMany = Sequences.Sequences.ZeroOrMany;
305
306         using (var scope = new TempLinksTestScope(useSequences: true))
307         {
308             var links = scope.Links;
309             var sequences = scope.Sequences;
310
311             var e1 = links.Create();
312             var e2 = links.Create();
313
314             var sequence = new[]
315             {
316                 e1, e2, e1, e2 // mama / papa
317             };
318
319             var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
320
321             var balancedVariant = balancedVariantConverter.Convert(sequence);
322
323             // 1: [1]
324             // 2: [2]
325             // 3: [1,2]
326             // 4: [1,2,1,2]
327
328             var doublet = links.GetSource(balancedVariant);
329
330             var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
331
332             Assert.True(matchedSequences1.Count == 0);
333
334             var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
335
336             Assert.True(matchedSequences2.Count == 0);
337
338             var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
339
340             Assert.True(matchedSequences3.Count == 0);
341
342             var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
343
344             Assert.Contains(doublet, matchedSequences4);
345             Assert.Contains(balancedVariant, matchedSequences4);
346
347             for (var i = 0; i < sequence.Length; i++)
348             {
349                 links.Delete(sequence[i]);
350             }
351         }
352     }
353
354     [Fact]
355     public static void IndexTest()
356     {
357         using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
358             ↳ true }, useSequences: true))
359         {
360             var links = scope.Links;
361             var sequences = scope.Sequences;
362             var index = sequences.Options.Index;
363
364             var e1 = links.Create();
365             var e2 = links.Create();
366
367             var sequence = new[]
368             {

```

```

368         e1, e2, e1, e2 // mama / papa
369     };
370
371     Assert.False(index.MightContain(sequence));
372
373     index.Add(sequence);
374
375     Assert.True(index.MightContain(sequence));
376 }
377 }
378
379 /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%
    ↳ DO%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87
    ↳ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
380 private static readonly string _exampleText =
381     @"([english
    ↳ version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
382
383 Обозначение пустоты, какое оно? Темнота ли это? Там где отсутствие света, отсутствие фотонов
    ↳ (носителей света)? Или это то, что полностью отражает свет? Пустой белый лист бумаги? Там
    ↳ где есть место для нового начала? Разве пустота это не характеристика пространства?
    ↳ Пространство это то, что можно чем-то наполнить?
384
385 [[чёрное пространство, белое
    ↳ пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png
    ↳ "чёрное пространство, белое пространство")](https://raw.githubusercontent.com/Konard/Links
    ↳ Platform/master/doc/Intro/1.png)
386
387 Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая
    ↳ форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?
388
389 [[чёрное пространство, чёрная
    ↳ точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png
    ↳ "чёрное пространство, чёрная
    ↳ точка")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png)
390
391 А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть
    ↳ так? Инверсия? Отражение? Сумма?
392
393 [[белая точка, чёрная
    ↳ точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png "белая
    ↳ точка, чёрная
    ↳ точка")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png)
394
395 А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет
    ↳ если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой?
    ↳ Гранью? Разделителем? Единицей?
396
397 [[две белые точки, чёрная вертикальная
    ↳ линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png "две
    ↳ белые точки, чёрная вертикальная
    ↳ линия")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)
398
399 Можно ли замкнуть движение? Может ли это быть кругом? Можно ли замкнуть время? Или остаётся
    ↳ только спираль? Но что если замкнуть предел? Создать ограничение, разделение? Получится
    ↳ замкнутая область? Полностью отделённая от всего остального? Но что это всё остальное? Что
    ↳ можно делить? В каком направлении? Ничего или всё? Пустота или полнота? Начало или конец?
    ↳ Или может быть это единица и ноль? Дуальность? Противоположность? А что будет с кругом если
    ↳ у него нет размера? Будет ли круг точкой? Точка состоящая из точек?
400
401 [[белая вертикальная линия, чёрный
    ↳ круг](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png "белая
    ↳ вертикальная линия, чёрный
    ↳ круг")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png)
402
403 Как ещё можно использовать грань, черту, линию? А что если она может что-то соединять, может
    ↳ тогда её нужно повернуть? Почему то, что перпендикулярно вертикальному горизонтально?
    ↳ Горизонт? Инвертирует ли это смысл? Что такое смысл? Из чего состоит смысл? Существует ли
    ↳ элементарная единица смысла?
404
405 [[белый круг, чёрная горизонтальная
    ↳ линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png "белый
    ↳ круг, чёрная горизонтальная
    ↳ линия")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png)
406
407 Соединять, допустим, а какой смысл в этом есть ещё? Что если помимо смысла "соединить,
    ↳ связать", есть ещё и смысл направления "от начала к концу"? От предка к потомку? От
    ↳ родителя к ребёнку? От общего к частному?
408

```

```

409 [![белая горизонтальная линия, чёрная горизонтальная
    ↳ стрелка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png
    ↳ "белая горизонтальная линия, чёрная горизонтальная
    ↳ стрелка")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png)
410
411 Шаг назад. Возьмём опять отделённую область, которая лишь та же замкнутая линия, что ещё она
    ↳ может представлять собой? Объект? Но в чём его суть? Разве не в том, что у него есть
    ↳ граница, разделяющая внутреннее и внешнее? Допустим связь, стрелка, линия соединяет два
    ↳ объекта, как бы это выглядело?
412
413 [![белая связь, чёрная направленная
    ↳ связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png "белая
    ↳ связь, чёрная направленная
    ↳ связь")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png)
414
415 Допустим у нас есть смысл "связать" и смысл "направления", много ли это нам даёт? Много ли
    ↳ вариантов интерпретации? А что если уточнить, каким именно образом выполнена связь? Что если
    ↳ можно задать ей чёткий, конкретный смысл? Что это будет? Тип? Глагол? Связка? Действие?
    ↳ Трансформация? Переход из состояния в состояние? Или всё это и есть объект, суть которого в
    ↳ его конечном состоянии, если конечно конец определён направлением?
416
417 [![белая обычная и направленная связи, чёрная типизированная
    ↳ связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png "белая
    ↳ обычная и направленная связи, чёрная типизированная
    ↳ связь")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png)
418
419 А что если всё это время, мы смотрели на суть как бы снаружи? Можно ли взглянуть на это изнутри?
    ↳ Что будет внутри объектов? Объекты ли это? Или это связи? Может ли эта структура описать
    ↳ сама себя? Но что тогда получится, разве это не рекурсия? Может это фрактал?
420
421 [![белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная
    ↳ связь с рекурсивной внутренней
    ↳ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png
    ↳ "белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с рекурсивной внутренней структурой")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png)
422
423 На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
    ↳ рекурсии или фрактала?
424
425 [![белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с двойной рекурсивной внутренней
    ↳ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png
    ↳ "белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с двойной рекурсивной внутренней структурой")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)
426
427 Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
    ↳ Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
428
429 [![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности,
    ↳ чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://
    ↳ raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png "белая обычная и
    ↳ направленная связи со структурой из 8 цветных элементов последовательности, чёрная
    ↳ типизированная связь со структурой из 8 цветных элементов последовательности")](https://raw
    ↳ .githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)
430
431 ...
432
433 [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animat
    ↳ ion-500.gif
    ↳ "анимация")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
    ↳ -animation-500.gif)";
434
435     private static readonly string _exampleLoremIpsumText =
436         @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
    ↳ incididunt ut labore et dolore magna aliqua.
437 Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
    ↳ consequat.";
438
439     [Fact]
440     public static void CompressionTest()
441     {
442         using (var scope = new TempLinksTestScope(useSequences: true))
443         {
444             var links = scope.Links;
445             var sequences = scope.Sequences;
446
447             var e1 = links.Create();

```



```

448     var e2 = links.Create();
449
450     var sequence = new[]
451     {
452         e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
453     };
454
455     var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
456     var totalSequenceSymbolFrequencyCounter = new
457         ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
458     var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
459         ↳ totalSequenceSymbolFrequencyCounter);
460     var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
461         ↳ balancedVariantConverter, doubletFrequenciesCache);
462
463     var compressedVariant = compressingConverter.Convert(sequence);
464
465     // 1: [1]          (1->1) point
466     // 2: [2]          (2->2) point
467     // 3: [1,2]        (1->2) doublet
468     // 4: [1,2,1,2]    (3->3) doublet
469
470     Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
471     Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
472     Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
473     Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
474
475     var source = _constants.SourcePart;
476     var target = _constants.TargetPart;
477
478     Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
479     Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
480     Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
481     Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
482
483     // 4 - length of sequence
484     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
485         ↳ == sequence[0]);
486     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
487         ↳ == sequence[1]);
488     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
489         ↳ == sequence[2]);
490     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
491         ↳ == sequence[3]);
492 }
493
494 [Fact]
495 public static void CompressionEfficiencyTest()
496 {
497     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
498         ↳ StringSplitOptions.RemoveEmptyEntries);
499     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
500     var totalCharacters = arrays.Select(x => x.Length).Sum();
501
502     using (var scope1 = new TempLinksTestScope(useSequences: true))
503     using (var scope2 = new TempLinksTestScope(useSequences: true))
504     using (var scope3 = new TempLinksTestScope(useSequences: true))
505     {
506         scope1.Links.Unsync.UseUnicode();
507         scope2.Links.Unsync.UseUnicode();
508         scope3.Links.Unsync.UseUnicode();
509
510         var balancedVariantConverter1 = new
511             ↳ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
512         var totalSequenceSymbolFrequencyCounter = new
513             ↳ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
514         var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
515             ↳ totalSequenceSymbolFrequencyCounter);
516         var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
517             ↳ balancedVariantConverter1, linkFrequenciesCache1,
518             ↳ doInitialFrequenciesIncrement: false);
519
520         //var compressor2 = scope2.Sequences;
521         var compressor3 = scope3.Sequences;
522
523         var constants = Default<LinksConstants<ulong>>.Instance;

```

```

513 var sequences = compressor3;
514 //var meaningRoot = links.CreatePoint();
515 //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
516 //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
517 //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
    ↳ constants.Itself);

518
519 //var unaryNumberToAddressConverter = new
    ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
520 //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
    ↳ unaryOne);
521 //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
    ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
522 //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
    ↳ frequencyPropertyMarker, frequencyMarker);
523 //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
    ↳ frequencyPropertyOperator, frequencyIncrementer);
524 //var linkToItsFrequencyNumberConverter = new
    ↳ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
    ↳ unaryNumberToAddressConverter);

525
526 var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
    ↳ totalSequenceSymbolFrequencyCounter);

527
528 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
    ↳ ncyNumberConverter<ulong>(linkFrequenciesCache3);

529
530 var sequenceToItsLocalElementLevelsConverter = new
    ↳ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
    ↳ linkToItsFrequencyNumberConverter);
531 var optimalVariantConverter = new
    ↳ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
    ↳ sequenceToItsLocalElementLevelsConverter);

532
533 var compressed1 = new ulong[arrays.Length];
534 var compressed2 = new ulong[arrays.Length];
535 var compressed3 = new ulong[arrays.Length];
536
537 var START = 0;
538 var END = arrays.Length;
539
540 //for (int i = START; i < END; i++)
541 //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
542
543 var initialCount1 = scope2.Links.Unsync.Count();
544
545 var sw1 = Stopwatch.StartNew();
546
547 for (int i = START; i < END; i++)
548 {
549     linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
550     compressed1[i] = compressor1.Convert(arrays[i]);
551 }
552
553 var elapsed1 = sw1.Elapsed;
554
555 var balancedVariantConverter2 = new
    ↳ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
556
557 var initialCount2 = scope2.Links.Unsync.Count();
558
559 var sw2 = Stopwatch.StartNew();
560
561 for (int i = START; i < END; i++)
562 {
563     compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
564 }
565
566 var elapsed2 = sw2.Elapsed;
567
568 for (int i = START; i < END; i++)
569 {
570     linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
571 }
572
573 var initialCount3 = scope3.Links.Unsync.Count();
574
575 var sw3 = Stopwatch.StartNew();
576

```

```

577 for (int i = START; i < END; i++)
578 {
579     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
580     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
581 }
582
583 var elapsed3 = sw3.Elapsed;
584
585 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
    ↳ Optimal variant: {elapsed3}");
586
587 // Assert.True(elapsed1 > elapsed2);
588
589 // Checks
590 for (int i = START; i < END; i++)
591 {
592     var sequence1 = compressed1[i];
593     var sequence2 = compressed2[i];
594     var sequence3 = compressed3[i];
595
596     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
    ↳ scope1.Links.Unsync);
597
598     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
    ↳ scope2.Links.Unsync);
599
600     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
    ↳ scope3.Links.Unsync);
601
602     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
    ↳ link.IsPartialPoint());
603     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
    ↳ link.IsPartialPoint());
604     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
    ↳ link.IsPartialPoint());
605
606     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
    ↳ arrays[i].Length > 3)
607     //    Assert.False(structure1 == structure2);
608     //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
    ↳ arrays[i].Length > 3)
609     //    Assert.False(structure3 == structure2);
610
611     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
612     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
613 }
614
615 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
    ↳ totalCharacters);
616 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
    ↳ totalCharacters);
617 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
    ↳ totalCharacters);
618
619 Console.WriteLine($"{{(double)(scope1.Links.Unsync.Count() - initialCount1) /
    ↳ totalCharacters}} | {{(double)(scope2.Links.Unsync.Count() - initialCount2) /
    ↳ totalCharacters}} | {{(double)(scope3.Links.Unsync.Count() - initialCount3) /
    ↳ totalCharacters}}");
620
621 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
    ↳ scope2.Links.Unsync.Count() - initialCount2);
622 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
    ↳ scope2.Links.Unsync.Count() - initialCount2);
623
624 var duplicateProvider1 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
625 var duplicateProvider2 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
626 var duplicateProvider3 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
627
628 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
629 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
630 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
631
632 var duplicates1 = duplicateCounter1.Count();
633

```

```

634     ConsoleHelpers.Debug("-----");
635
636     var duplicates2 = duplicateCounter2.Count();
637
638     ConsoleHelpers.Debug("-----");
639
640     var duplicates3 = duplicateCounter3.Count();
641
642     Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
643
644     linkFrequenciesCache1.ValidateFrequencies();
645     linkFrequenciesCache3.ValidateFrequencies();
646 }
647
648
649 [Fact]
650 public static void CompressionStabilityTest()
651 {
652     // TODO: Fix bug (do a separate test)
653     //const ulong minNumbers = 0;
654     //const ulong maxNumbers = 1000;
655
656     const ulong minNumbers = 10000;
657     const ulong maxNumbers = 12500;
658
659     var strings = new List<string>();
660
661     for (ulong i = minNumbers; i < maxNumbers; i++)
662     {
663         strings.Add(i.ToString());
664     }
665
666     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
667     var totalCharacters = arrays.Select(x => x.Length).Sum();
668
669     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
        ↪ SequencesOptions<ulong> { UseCompression = true,
        ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
670     using (var scope2 = new TempLinksTestScope(useSequences: true))
671     {
672         scope1.Links.UseUnicode();
673         scope2.Links.UseUnicode();
674
675         //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
676         var compressor1 = scope1.Sequences;
677         var compressor2 = scope2.Sequences;
678
679         var compressed1 = new ulong[arrays.Length];
680         var compressed2 = new ulong[arrays.Length];
681
682         var sw1 = Stopwatch.StartNew();
683
684         var START = 0;
685         var END = arrays.Length;
686
687         // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
688         // Stability issue starts at 10001 or 11000
689         //for (int i = START; i < END; i++)
690         //{
691             // var first = compressor1.Compress(arrays[i]);
692             // var second = compressor1.Compress(arrays[i]);
693
694             // if (first == second)
695                 // compressed1[i] = first;
696             // else
697                 // {
698                     // // TODO: Find a solution for this case
699                 // }
700         //}
701
702         for (int i = START; i < END; i++)
703         {
704             var first = compressor1.Create(arrays[i].ShiftRight());
705             var second = compressor1.Create(arrays[i].ShiftRight());
706
707             if (first == second)
708             {
709                 compressed1[i] = first;
710             }
711             else

```

```

712     {
713         // TODO: Find a solution for this case
714     }
715 }
716
717 var elapsed1 = sw1.Elapsed;
718
719 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
720
721 var sw2 = Stopwatch.StartNew();
722
723 for (int i = START; i < END; i++)
724 {
725     var first = balancedVariantConverter.Convert(arrays[i]);
726     var second = balancedVariantConverter.Convert(arrays[i]);
727
728     if (first == second)
729     {
730         compressed2[i] = first;
731     }
732 }
733
734 var elapsed2 = sw2.Elapsed;
735
736 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
737     ↳ {elapsed2}");
738
739 Assert.True(elapsed1 > elapsed2);
740
741 // Checks
742 for (int i = START; i < END; i++)
743 {
744     var sequence1 = compressed1[i];
745     var sequence2 = compressed2[i];
746
747     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
748     {
749         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
750             ↳ scope1.Links);
751
752         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
753             ↳ scope2.Links);
754
755         //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
756             ↳ link.IsPartialPoint());
757         //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
758             ↳ link.IsPartialPoint());
759
760         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
761             ↳ arrays[i].Length > 3)
762             // Assert.False(structure1 == structure2);
763
764         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
765     }
766 }
767
768 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
769 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
770
771 Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
772     ↳ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
773     ↳ totalCharacters}}");
774
775 Assert.True(scope1.Links.Count() <= scope2.Links.Count());
776
777 //compressor1.ValidateFrequencies();
778 }
779
780 [Fact]
781 public static void RandomNumbersCompressionQualityTest()
782 {
783     const ulong N = 500;
784
785     //const ulong minNumbers = 10000;
786     //const ulong maxNumbers = 20000;
787
788     //var strings = new List<string>();

```

```

783 //for (ulong i = 0; i < N; i++)
784 //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
    ↪ maxNumbers).ToString());
785
786 var strings = new List<string>();
787
788 for (ulong i = 0; i < N; i++)
789 {
790     strings.Add(RandomHelpers.Default.NextUInt64().ToString());
791 }
792
793 strings = strings.Distinct().ToList();
794
795 var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
796 var totalCharacters = arrays.Select(x => x.Length).Sum();
797
798 using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
    ↪ SequencesOptions<ulong> { UseCompression = true,
    ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
799 using (var scope2 = new TempLinksTestScope(useSequences: true))
800 {
801     scope1.Links.UseUnicode();
802     scope2.Links.UseUnicode();
803
804     var compressor1 = scope1.Sequences;
805     var compressor2 = scope2.Sequences;
806
807     var compressed1 = new ulong[arrays.Length];
808     var compressed2 = new ulong[arrays.Length];
809
810     var sw1 = Stopwatch.StartNew();
811
812     var START = 0;
813     var END = arrays.Length;
814
815     for (int i = START; i < END; i++)
816     {
817         compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
818     }
819
820     var elapsed1 = sw1.Elapsed;
821
822     var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
823
824     var sw2 = Stopwatch.StartNew();
825
826     for (int i = START; i < END; i++)
827     {
828         compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
829     }
830
831     var elapsed2 = sw2.Elapsed;
832
833     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
    ↪ {elapsed2}");
834
835     Assert.True(elapsed1 > elapsed2);
836
837     // Checks
838     for (int i = START; i < END; i++)
839     {
840         var sequence1 = compressed1[i];
841         var sequence2 = compressed2[i];
842
843         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
844         {
845             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
    ↪ scope1.Links);
846
847             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
    ↪ scope2.Links);
848
849             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
850         }
851     }
852
853     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
854     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
855

```

```

856         Debug.WriteLine($"{((double)(scope1.Links.Count() - UnicodeMap.MapSize) /
      ↳ totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
      ↳ totalCharacters}");
857
858         // Can be worse than balanced variant
859         //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
860
861         //compressor1.ValidateFrequencies();
862     }
863 }
864
865 [Fact]
866 public static void AllTreeBreakDownAtSequencesCreationBugTest()
867 {
868     // Made out of AllPossibleConnectionsTest test.
869
870     //const long sequenceLength = 5; //100% bug
871     const long sequenceLength = 4; //100% bug
872     //const long sequenceLength = 3; //100% _no_bug (ok)
873
874     using (var scope = new TempLinksTestScope(useSequences: true))
875     {
876         var links = scope.Links;
877         var sequences = scope.Sequences;
878
879         var sequence = new ulong[sequenceLength];
880         for (var i = 0; i < sequenceLength; i++)
881         {
882             sequence[i] = links.Create();
883         }
884
885         var createResults = sequences.CreateAllVariants2(sequence);
886         Global.Trash = createResults;
887
888         for (var i = 0; i < sequenceLength; i++)
889         {
890             links.Delete(sequence[i]);
891         }
892     }
893 }
894
895 [Fact]
896 public static void AllPossibleConnectionsTest()
897 {
898     const long sequenceLength = 5;
899
900     using (var scope = new TempLinksTestScope(useSequences: true))
901     {
902         var links = scope.Links;
903         var sequences = scope.Sequences;
904
905         var sequence = new ulong[sequenceLength];
906         for (var i = 0; i < sequenceLength; i++)
907         {
908             sequence[i] = links.Create();
909         }
910
911         var createResults = sequences.CreateAllVariants2(sequence);
912         var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
913
914         for (var i = 0; i < 1; i++)
915         {
916             var sw1 = Stopwatch.StartNew();
917             var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
918
919             var sw2 = Stopwatch.StartNew();
920             var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
921
922             var sw3 = Stopwatch.StartNew();
923             var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
924
925             var sw4 = Stopwatch.StartNew();
926             var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
927
928             Global.Trash = searchResults3;
929             Global.Trash = searchResults4; //-V3008
930
931             var intersection1 = createResults.Intersect(searchResults1).ToList();
932             Assert.True(intersection1.Count == createResults.Length);
933

```

```

934         var intersection2 = reverseResults.Intersect(searchResults1).ToList();
935         Assert.True(intersection2.Count == reverseResults.Length);
936
937         var intersection0 = searchResults1.Intersect(searchResults2).ToList();
938         Assert.True(intersection0.Count == searchResults2.Count);
939
940         var intersection3 = searchResults2.Intersect(searchResults3).ToList();
941         Assert.True(intersection3.Count == searchResults3.Count);
942
943         var intersection4 = searchResults3.Intersect(searchResults4).ToList();
944         Assert.True(intersection4.Count == searchResults4.Count);
945     }
946 }
947
948 for (var i = 0; i < sequenceLength; i++)
949 {
950     links.Delete(sequence[i]);
951 }
952 }
953 }
954
955 [Fact(Skip = "Correct implementation is pending")]
956 public static void CalculateAllUsagesTest()
957 {
958     const long sequenceLength = 3;
959
960     using (var scope = new TempLinksTestScope(useSequences: true))
961     {
962         var links = scope.Links;
963         var sequences = scope.Sequences;
964
965         var sequence = new ulong[sequenceLength];
966         for (var i = 0; i < sequenceLength; i++)
967         {
968             sequence[i] = links.Create();
969         }
970
971         var createResults = sequences.CreateAllVariants2(sequence);
972
973         //var reverseResults =
974         ↪ sequences.CreateAllVariants2(sequence.Reverse().ToArray());
975
976         for (var i = 0; i < 1; i++)
977         {
978             var linksTotalUsages1 = new ulong[links.Count() + 1];
979
980             sequences.CalculateAllUsages(linksTotalUsages1);
981
982             var linksTotalUsages2 = new ulong[links.Count() + 1];
983
984             sequences.CalculateAllUsages2(linksTotalUsages2);
985
986             var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
987             Assert.True(intersection1.Count == linksTotalUsages2.Length);
988         }
989
990         for (var i = 0; i < sequenceLength; i++)
991         {
992             links.Delete(sequence[i]);
993         }
994     }
995 }
996 }

```

1.186 ./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 constants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
51     using (var dataMemory = new HeapResizableDirectMemory())
52     using (var indexMemory = new HeapResizableDirectMemory())
53     using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory,
    ↪ SplitMemoryLinks<TLink>.DefaultLinksSizeStep, constants))
54     {
55         action(memory);
56     }
57 }
58 }
59 }

```

1.187 ./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,
        ↪ UInt32SplitMemoryLinks.DefaultLinksSizeStep, constants))
45         {
46             action(memory);
47         }
48     }
49 }
50 }

```

1.188 ./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().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 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,
        ↪ UInt64SplitMemoryLinks.DefaultLinksSizeStep, constants))
45             {
46                 action(memory);
47             }
48         }
49     }
50 }

```

```

48     }
49 }
50 }

```

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

```

1  using System.IO;
2  using Platform.Disposables;
3  using Platform.Data.Doublets.Sequences;
4  using Platform.Data.Doublets.Decorators;
5  using Platform.Data.Doublets.Memory.United.Specific;
6  using Platform.Data.Doublets.Memory.Split.Specific;
7  using Platform.Memory;
8
9  namespace Platform.Data.Doublets.Tests
10 {
11     public class TempLinksTestScope : DisposableBase
12     {
13         public ILinks<ulong> MemoryAdapter { get; }
14         public SynchronizedLinks<ulong> Links { get; }
15         public Sequences.Sequences Sequences { get; }
16         public string TempFilename { get; }
17         public string TempTransactionLogFilename { get; }
18         private readonly bool _deleteFiles;
19
20         public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
            ↪ useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
            ↪ useLog) { }
21
22         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
            ↪ true, bool useSequences = false, bool useLog = false)
23         {
24             _deleteFiles = deleteFiles;
25             TempFilename = Path.GetTempFileName();
26             TempTransactionLogFilename = Path.GetTempFileName();
27             //var coreMemoryAdapter = new UInt64UnitedMemoryLinks(TempFilename);
28             var coreMemoryAdapter = new UInt64SplitMemoryLinks(new
                ↪ FileMappedResizableDirectMemory(TempFilename), new
                ↪ FileMappedResizableDirectMemory(Path.ChangeExtension(TempFilename, "indexes")),
                ↪ UInt64SplitMemoryLinks.DefaultLinksSizeStep, new LinksConstants<ulong>(),
                ↪ Memory.IndexTreeType.Default, useLinkedList: true);
29             MemoryAdapter = useLog ? (ILinks<ulong>)new
                ↪ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
                ↪ coreMemoryAdapter;
30             Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
31             if (useSequences)
32             {
33                 Sequences = new Sequences.Sequences(Links, sequencesOptions);
34             }
35         }
36
37         protected override void Dispose(bool manual, bool wasDisposed)
38         {
39             if (!wasDisposed)
40             {
41                 Links.Unsync.DisposeIfPossible();
42                 if (_deleteFiles)
43                 {
44                     DeleteFiles();
45                 }
46             }
47         }
48
49         public void DeleteFiles()
50         {
51             File.Delete(TempFilename);
52             File.Delete(TempTransactionLogFilename);
53         }
54     }
55 }

```

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

```

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.IO;
5  using System.Text;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Xunit;
9  using Platform.Disposables;
10 using Platform.Ranges;
11 using Platform.Random;
12 using Platform.Timestamps;
13 using Platform.Reflection;
14 using Platform.Singletons;
15 using Platform.Scopes;
16 using Platform.Counters;
17 using Platform.Diagnostics;
18 using Platform.IO;
19 using Platform.Memory;
20 using Platform.Data.Doublets.Decorators;
21 using Platform.Data.Doublets.Memory.United.Specific;
22
23 namespace Platform.Data.Doublets.Tests
24 {
25     public static class UInt64LinksTests
26     {
27         private static readonly LinksConstants<ulong> _constants =
28             ↪ Default<LinksConstants<ulong>>.Instance;
29
30         private const long Iterations = 10 * 1024;
31
32         #region Concept
33
34         [Fact]
35         public static void MultipleCreateAndDeleteTest()
36         {
37             using (var scope = new Scope<Types<HeapResizableDirectMemory,
38                 ↪ UInt64UnitedMemoryLinks>>())
39             {

```

```

38         new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeletions(100);
39     }
40 }
41
42 [Fact]
43 public static void CascadeUpdateTest()
44 {
45     var itself = _constants.Itself;
46     using (var scope = new TempLinksTestScope(useLog: true))
47     {
48         var links = scope.Links;
49
50         var l1 = links.Create();
51         var l2 = links.Create();
52
53         l2 = links.Update(l2, l2, l1, l2);
54
55         links.CreateAndUpdate(l2, itself);
56         links.CreateAndUpdate(l2, itself);
57
58         l2 = links.Update(l2, l1);
59
60         links.Delete(l2);
61
62         Global.Trash = links.Count();
63
64         links.Unsync.DisposeIfPossible(); // Close links to access log
65
66         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
67     }
68 }
69
70 [Fact]
71 public static void BasicTransactionLogTest()
72 {
73     using (var scope = new TempLinksTestScope(useLog: true))
74     {
75         var links = scope.Links;
76         var l1 = links.Create();
77         var l2 = links.Create();
78
79         Global.Trash = links.Update(l2, l2, l1, l2);
80
81         links.Delete(l1);
82
83         links.Unsync.DisposeIfPossible(); // Close links to access log
84
85         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
86     }
87 }
88
89 [Fact]
90 public static void TransactionAutoRevertedTest()
91 {
92     // Auto Reverted (Because no commit at transaction)
93     using (var scope = new TempLinksTestScope(useLog: true))
94     {
95         var links = scope.Links;
96         var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
97         using (var transaction = transactionsLayer.BeginTransaction())
98         {
99             var l1 = links.Create();
100             var l2 = links.Create();
101
102             links.Update(l2, l2, l1, l2);
103         }
104
105         Assert.Equal(0UL, links.Count());
106
107         links.Unsync.DisposeIfPossible();
108
109         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
110         Assert.Single(transitions);
111     }
112 }

```

```

113 [Fact]
114 public static void TransactionUserCodeErrorNoDataSavedTest()
115 {
116     // User Code Error (Autoreverted), no data saved
117     var itself = _constants.Itself;
118
119     TempLinksTestScope lastScope = null;
120     try
121     {
122         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
123             ↪ useLog: true))
124         {
125             var links = scope.Links;
126             var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor_
127             ↪ atorBase<ulong>)links.Unsync).Links;
128             using (var transaction = transactionsLayer.BeginTransaction())
129             {
130                 var l1 = links.CreateAndUpdate(itself, itself);
131                 var l2 = links.CreateAndUpdate(itself, itself);
132
133                 l2 = links.Update(l2, l2, l1, l2);
134
135                 links.CreateAndUpdate(l2, itself);
136                 links.CreateAndUpdate(l2, itself);
137
138                 //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi_
139                 ↪ tion>(scope.TempTransactionLogFilename);
140
141                 l2 = links.Update(l2, l1);
142
143                 links.Delete(l2);
144
145                 ExceptionThrower();
146
147                 transaction.Commit();
148             }
149             Global.Trash = links.Count();
150         }
151     } catch
152     {
153         Assert.False(lastScope == null);
154
155         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l_
156         ↪ astScope.TempTransactionLogFilename);
157
158         Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
159         ↪ transitions[0].After.IsNull());
160
161         lastScope.DeleteFiles();
162     }
163 }
164
165 [Fact]
166 public static void TransactionUserCodeErrorSomeDataSavedTest()
167 {
168     // User Code Error (Autoreverted), some data saved
169     var itself = _constants.Itself;
170
171     TempLinksTestScope lastScope = null;
172     try
173     {
174         using (var scope = new TempLinksTestScope(useLog: true))
175         {
176             var links = scope.Links;
177             l1 = links.CreateAndUpdate(itself, itself);
178             l2 = links.CreateAndUpdate(itself, itself);
179
180             l2 = links.Update(l2, l2, l1, l2);
181
182             links.CreateAndUpdate(l2, itself);
183             links.CreateAndUpdate(l2, itself);
184
185             links.Unsync.DisposeIfPossible();
186
187

```



```

188         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
189             ↪ scope.TempTransactionLogFilename);
190     }
191     using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
192         ↪ useLog: true))
193     {
194         var links = scope.Links;
195         var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
196         using (var transaction = transactionsLayer.BeginTransaction())
197         {
198             l2 = links.Update(l2, l1);
199             links.Delete(l2);
200             ExceptionThrower();
201             transaction.Commit();
202         }
203         Global.Trash = links.Count();
204     }
205 }
206 }
207 }
208 }
209 catch
210 {
211     Assert.False(lastScope == null);
212     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
213         ↪ Scope.TempTransactionLogFilename);
214     lastScope.DeleteFiles();
215 }
216 }
217 }
218 }
219 [Fact]
220 public static void TransactionCommit()
221 {
222     var itself = _constants.Itself;
223     var tempDatabaseFilename = Path.GetTempFileName();
224     var tempTransactionLogFilename = Path.GetTempFileName();
225     // Commit
226     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
227         ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
228     using (var links = new UInt64Links(memoryAdapter))
229     {
230         using (var transaction = memoryAdapter.BeginTransaction())
231         {
232             var l1 = links.CreateAndUpdate(itself, itself);
233             var l2 = links.CreateAndUpdate(itself, itself);
234             Global.Trash = links.Update(l2, l2, l1, l2);
235             links.Delete(l1);
236             transaction.Commit();
237         }
238         Global.Trash = links.Count();
239     }
240     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
241         ↪ sactionLogFilename);
242 }
243 }
244 }
245 }
246 }
247 }
248 }
249 [Fact]
250 public static void TransactionDamage()
251 {
252     var itself = _constants.Itself;
253     var tempDatabaseFilename = Path.GetTempFileName();
254     var tempTransactionLogFilename = Path.GetTempFileName();
255     // Commit
256     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
257         ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
258     using (var links = new UInt64Links(memoryAdapter))
259     {
260

```

```

261     using (var transaction = memoryAdapter.BeginTransaction())
262     {
263         var l1 = links.CreateAndUpdate(itself, itself);
264         var l2 = links.CreateAndUpdate(itself, itself);
265
266         Global.Trash = links.Update(l2, l2, l1, l2);
267
268         links.Delete(l1);
269
270         transaction.Commit();
271     }
272
273     Global.Trash = links.Count();
274 }
275
276 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
    ↪ sactionLogFilename);
277
278 // Damage database
279
280 FileHelpers.WriteFirst(tempTransactionLogFilename, new
    ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
281
282 // Try load damaged database
283 try
284 {
285     // TODO: Fix
286     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
    ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
287     using (var links = new UInt64Links(memoryAdapter))
288     {
289         Global.Trash = links.Count();
290     }
291 }
292 catch (NotSupportedException ex)
293 {
294     Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
    ↪ yet.");
295 }
296
297 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
    ↪ sactionLogFilename);
298
299 File.Delete(tempDatabaseFilename);
300 File.Delete(tempTransactionLogFilename);
301 }
302
303 [Fact]
304 public static void Bug1Test()
305 {
306     var tempDatabaseFilename = Path.GetTempFileName();
307     var tempTransactionLogFilename = Path.GetTempFileName();
308
309     var itself = _constants.Itself;
310
311     // User Code Error (Autoreverted), some data saved
312     try
313     {
314         ulong l1;
315         ulong l2;
316
317         using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
318         using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
    ↪ tempTransactionLogFilename))
319         using (var links = new UInt64Links(memoryAdapter))
320         {
321             l1 = links.CreateAndUpdate(itself, itself);
322             l2 = links.CreateAndUpdate(itself, itself);
323
324             l2 = links.Update(l2, l2, l1, l2);
325
326             links.CreateAndUpdate(l2, itself);
327             links.CreateAndUpdate(l2, itself);
328         }
329
330         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp_
    ↪ TransactionLogFilename);
331
332         using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))

```

```

333     using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
334         ↪ tempTransactionLogFilename))
335     using (var links = new UInt64Links(memoryAdapter))
336     {
337         using (var transaction = memoryAdapter.BeginTransaction())
338         {
339             l2 = links.Update(l2, l1);
340             links.Delete(l2);
341             ExceptionThrower();
342             transaction.Commit();
343         }
344         Global.Trash = links.Count();
345     }
346 }
347 }
348 }
349 }
350 catch
351 {
352     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
353         ↪ TransactionLogFilename);
354 }
355 File.Delete(tempDatabaseFilename);
356 File.Delete(tempTransactionLogFilename);
357 }
358
359 private static void ExceptionThrower() => throw new InvalidOperationException();
360
361 [Fact]
362 public static void PathsTest()
363 {
364     var source = _constants.SourcePart;
365     var target = _constants.TargetPart;
366
367     using (var scope = new TempLinksTestScope())
368     {
369         var links = scope.Links;
370         var l1 = links.CreatePoint();
371         var l2 = links.CreatePoint();
372
373         var r1 = links.GetByKeys(l1, source, target, source);
374         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
375     }
376 }
377
378 [Fact]
379 public static void RecursiveStringFormattingTest()
380 {
381     using (var scope = new TempLinksTestScope(useSequences: true))
382     {
383         var links = scope.Links;
384         var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
385
386         var a = links.CreatePoint();
387         var b = links.CreatePoint();
388         var c = links.CreatePoint();
389
390         var ab = links.GetOrCreate(a, b);
391         var cb = links.GetOrCreate(c, b);
392         var ac = links.GetOrCreate(a, c);
393
394         a = links.Update(a, c, b);
395         b = links.Update(b, a, c);
396         c = links.Update(c, a, b);
397
398         Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
399         Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
400         Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
401
402         Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
403             ↪ "(5:(4:5 (6:5 4)) 6)");
404         Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
405             ↪ "(6:(5:(4:5 6) 6) 4)");
406         Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
407             ↪ "(4:(5:4 (6:5 4)) 6)");
408     }
409 }

```

```

406 // TODO: Think how to build balanced syntax tree while formatting structure (eg.
407 ↪ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)"
408
409 Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
410 ↪ "{5}{5}{4}{6}");
411 Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
412 ↪ "{5}{6}{6}{4}");
413 Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
414 ↪ "{4}{5}{4}{6}");
415
416 }
417
418 private static void DefaultFormatter(StringBuilder sb, ulong link)
419 {
420     sb.Append(link.ToString());
421 }
422
423 #endregion
424
425 #region Performance
426
427 /*
428 public static void RunAllPerformanceTests()
429 {
430     try
431     {
432         links.TestLinksInSteps();
433     }
434     catch (Exception ex)
435     {
436         ex.WriteToConsole();
437     }
438     return;
439 }
440
441 try
442 {
443     //ThreadPool.SetMaxThreads(2, 2);
444
445     // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
446     ↪ результат
447     // Также это дополнительно помогает в отладке
448     // Увеличивает вероятность попадания информации в кэши
449     for (var i = 0; i < 10; i++)
450     {
451         //0 - 10 ГБ
452         //Каждые 100 МБ срез цифр
453
454         //links.TestGetSourceFunction();
455         //links.TestGetSourceFunctionInParallel();
456         //links.TestGetTargetFunction();
457         //links.TestGetTargetFunctionInParallel();
458         links.Create64BillionLinks();
459
460         links.TestRandomSearchFixed();
461         //links.Create64BillionLinksInParallel();
462         links.TestEachFunction();
463         //links.TestForeach();
464         //links.TestParallelForeach();
465     }
466
467     links.TestDeletionOfAllLinks();
468 }
469 catch (Exception ex)
470 {
471     ex.WriteToConsole();
472 }
473 }*/
474
475 /*
476 public static void TestLinksInSteps()
477 {
478     const long gibibyte = 1024 * 1024 * 1024;
479     const long mebibyte = 1024 * 1024;
480
481     var totalLinksToCreate = gibibyte /
482     ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;

```

```

478         var linksStep = 102 * mebibyte /
↳ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;

479
480         var creationMeasurements = new List<TimeSpan>();
481         var searchMeasurements = new List<TimeSpan>();
482         var deletionMeasurements = new List<TimeSpan>();
483
484         GetBaseRandomLoopOverhead(linksStep);
485         GetBaseRandomLoopOverhead(linksStep);
486
487         var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
488
489         ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
490
491         var loops = totalLinksToCreate / linksStep;
492
493         for (int i = 0; i < loops; i++)
494         {
495             creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
496             searchMeasurements.Add(Measure(() => links.RunRandomSearches(linksStep)));
497
498             Console.WriteLine("\rC + S {0}/{1}", i + 1, loops);
499         }
500
501         ConsoleHelpers.Debug();
502
503         for (int i = 0; i < loops; i++)
504         {
505             deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
506
507             Console.WriteLine("\rD {0}/{1}", i + 1, loops);
508         }
509
510         ConsoleHelpers.Debug();
511
512         ConsoleHelpers.Debug("C S D");
513
514         for (int i = 0; i < loops; i++)
515         {
516             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
↳ searchMeasurements[i], deletionMeasurements[i]);
517         }
518
519         ConsoleHelpers.Debug("C S D (no overhead)");
520
521         for (int i = 0; i < loops; i++)
522         {
523             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
↳ searchMeasurements[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
524         }
525
526         ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
↳ links.Total);
527     }
528
529     private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
↳ amountToCreate)
530     {
531         for (long i = 0; i < amountToCreate; i++)
532             links.Create(0, 0);
533     }
534
535     private static TimeSpan GetBaseRandomLoopOverhead(long loops)
536     {
537         return Measure(() =>
538         {
539             ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
540             ulong result = 0;
541             for (long i = 0; i < loops; i++)
542             {
543                 var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
544                 var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
545
546                 result += maxValue + source + target;
547             }
548             Global.Trash = result;
549         });
550     }
551     */

```

```

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

```

```

627     var links = scope.Links;
628     ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
        ↪ Iterations);
629
630     ulong counter = 0;
631
632     //var firstLink = links.First();
633     var firstLink = links.Create();
634
635     var sw = Stopwatch.StartNew();
636
637     for (ulong i = 0; i < Iterations; i++)
638     {
639         counter += links.GetTarget(firstLink);
640     }
641
642     var elapsedTime = sw.Elapsed;
643
644     var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
645
646     links.Delete(firstLink);
647
648     ConsoleHelpers.Debug(
649         "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↪ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
650
651     }
652 }
653
654 [Fact(Skip = "performance test")]
655 public static void TestGetTargetInParallel()
656 {
657     using (var scope = new TempLinksTestScope())
658     {
659         var links = scope.Links;
660         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
        ↪ parallel.", Iterations);
661
662         long counter = 0;
663
664         //var firstLink = links.First();
665         var firstLink = links.Create();
666
667         var sw = Stopwatch.StartNew();
668
669         Parallel.For(0, Iterations, x =>
670         {
671             Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
672             //Interlocked.Increment(ref counter);
673         });
674
675         var elapsedTime = sw.Elapsed;
676
677         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
678
679         links.Delete(firstLink);
680
681         ConsoleHelpers.Debug(
682             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↪ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
683     }
684 }
685
686 // TODO: Заполнить базу данных перед тестом
687 /*
688 [Fact]
689 public void TestRandomSearchFixed()
690 {
691     var tempFilename = Path.GetTempFileName();
692
693     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
        ↪ DefaultLinksSizeStep))
694     {
695         long iterations = 64 * 1024 * 1024 /
        ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
696
697         ulong counter = 0;
698         var maxLink = links.Total;
699
700

```

```

701         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
702
703         var sw = Stopwatch.StartNew();
704
705         for (var i = iterations; i > 0; i--)
706         {
707             var source =
↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
708             var target =
↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
709
710             counter += links.Search(source, target);
711         }
712
713         var elapsedTime = sw.Elapsed;
714
715         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
716
717         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
↪ counter);
718     }
719
720     File.Delete(tempFilename);
721 }*/
722
723 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
724 public static void TestRandomSearchAll()
725 {
726     using (var scope = new TempLinksTestScope())
727     {
728         var links = scope.Links;
729         ulong counter = 0;
730
731         var maxLink = links.Count();
732
733         var iterations = links.Count();
734
735         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
↪ links.Count());
736
737         var sw = Stopwatch.StartNew();
738
739         for (var i = iterations; i > 0; i--)
740         {
741             var linksAddressRange = new
↪ Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);
742
743             var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
744             var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
745
746             counter += links.SearchOrDefault(source, target);
747         }
748
749         var elapsedTime = sw.Elapsed;
750
751         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
752
753         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪ Iterations per second), c: {3}",
↪ iterations, elapsedTime, (long)iterationsPerSecond, counter);
754     }
755 }
756
757 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
758 public static void TestEach()
759 {
760     using (var scope = new TempLinksTestScope())
761     {
762         var links = scope.Links;
763
764         var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
765
766         ConsoleHelpers.Debug("Testing Each function.");
767
768         var sw = Stopwatch.StartNew();
769
770         links.Each(counter.IncrementAndReturnTrue);
771
772         var elapsedTime = sw.Elapsed;
773

```



```

774
775         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
776
777         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
↪         links per second)",
778             counter, elapsedTime, (long)linksPerSecond);
779     }
780 }
781
782 /*
783 [Fact]
784 public static void TestForeach()
785 {
786     var tempFilename = Path.GetTempFileName();
787
788     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪     DefaultLinksSizeStep))
789     {
790         ulong counter = 0;
791
792         ConsoleHelpers.Debug("Testing foreach through links.");
793
794         var sw = Stopwatch.StartNew();
795
796         //foreach (var link in links)
797         //{
798             counter++;
799         //}
800
801         var elapsedTime = sw.Elapsed;
802
803         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
804
805         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
↪     links per second)", counter, elapsedTime, (long)linksPerSecond);
806     }
807
808     File.Delete(tempFilename);
809 }
810 */
811
812 /*
813 [Fact]
814 public static void TestParallelForeach()
815 {
816     var tempFilename = Path.GetTempFileName();
817
818     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪     DefaultLinksSizeStep))
819     {
820         long counter = 0;
821
822         ConsoleHelpers.Debug("Testing parallel foreach through links.");
823
824         var sw = Stopwatch.StartNew();
825
826         //Parallel.ForEach((IEnumerable<ulong>)links, x =>
827         //{
828             Interlocked.Increment(ref counter);
829         //});
830
831         var elapsedTime = sw.Elapsed;
832
833         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
834
835         ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↪     {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
836     }
837
838     File.Delete(tempFilename);
839 }
840 */
841
842 [Fact(Skip = "performance test")]
843 public static void Create64BillionLinks()
844 {
845     using (var scope = new TempLinksTestScope())
846     {
847

```

```

848     var links = scope.Links;
849     var linksBeforeTest = links.Count();
850
851     long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;
852
853     ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
854
855     var elapsedTime = Performance.Measure(() =>
856     {
857         for (long i = 0; i < linksToCreate; i++)
858         {
859             links.Create();
860         }
861     });
862
863     var linksCreated = links.Count() - linksBeforeTest;
864     var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
865
866     ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
867
868     ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
869         ↪ linksCreated, elapsedTime,
870         (long)linksPerSecond);
871 }
872
873 [Fact(Skip = "performance test")]
874 public static void Create64BillionLinksInParallel()
875 {
876     using (var scope = new TempLinksTestScope())
877     {
878         var links = scope.Links;
879         var linksBeforeTest = links.Count();
880
881         var sw = Stopwatch.StartNew();
882
883         long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;
884
885         ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
886
887         Parallel.For(0, linksToCreate, x => links.Create());
888
889         var elapsedTime = sw.Elapsed;
890
891         var linksCreated = links.Count() - linksBeforeTest;
892         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
893
894         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
895             ↪ linksCreated, elapsedTime,
896             (long)linksPerSecond);
897     }
898
899 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
900 public static void TestDeletionOfAllLinks()
901 {
902     using (var scope = new TempLinksTestScope())
903     {
904         var links = scope.Links;
905         var linksBeforeTest = links.Count();
906
907         ConsoleHelpers.Debug("Deleting all links");
908
909         var elapsedTime = Performance.Measure(links.DeleteAll);
910
911         var linksDeleted = linksBeforeTest - links.Count();
912         var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
913
914         ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
915             ↪ linksDeleted, elapsedTime,
916             (long)linksPerSecond);
917     }
918 }
919 #endregion
920 }
921 }

```



```

29         var fromUnaryNumberConverterUsingAddOperation = new
        ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
30     for (int i = 0; i < N; i++)
31     {
32         Assert.Equal(numbers[i],
        ↪ fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
33         Assert.Equal(numbers[i],
        ↪ fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
34     }
35 }
36 }
37 }
38 }

```

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

```

1  using Xunit;
2  using Platform.Converters;
3  using Platform.Memory;
4  using Platform.Reflection;
5  using Platform.Scopes;
6  using Platform.Data.Numbers.Raw;
7  using Platform.Data.Doublets.Incremeters;
8  using Platform.Data.Doublets.Numbers.Unary;
9  using Platform.Data.Doublets.PropertyOperators;
10 using Platform.Data.Doublets.Sequences.Converters;
11 using Platform.Data.Doublets.Sequences.Indexes;
12 using Platform.Data.Doublets.Sequences.Walkers;
13 using Platform.Data.Doublets.Unicode;
14 using Platform.Data.Doublets.Memory.United.Generic;
15 using Platform.Data.Doublets.CriterionMatchers;
16
17 namespace Platform.Data.Doublets.Tests
18 {
19     public static class UnicodeConvertersTests
20     {
21         [Fact]
22         public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
23         {
24             using (var scope = new TempLinksTestScope())
25             {
26                 var links = scope.Links;
27                 var meaningRoot = links.CreatePoint();
28                 var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
29                 var powerOf2ToUnaryNumberConverter = new
        ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
30                 var addressToUnaryNumberConverter = new
        ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
31                 var unaryNumberToAddressConverter = new
        ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
        ↪ powerOf2ToUnaryNumberConverter);
32                 TestCharAndUnicodeSymbolConverters(links, meaningRoot,
        ↪ addressToUnaryNumberConverter, unaryNumberToAddressConverter);
33             }
34         }
35
36         [Fact]
37         public static void CharAndRawNumberUnicodeSymbolConvertersTest()
38         {
39             using (var scope = new Scope<Types<HeapResizableDirectMemory,
        ↪ UnitedMemoryLinks<ulong>>>())
40             {
41                 var links = scope.Use<ILinks<ulong>>>();
42                 var meaningRoot = links.CreatePoint();
43                 var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
44                 var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
45                 TestCharAndUnicodeSymbolConverters(links, meaningRoot,
        ↪ addressToRawNumberConverter, rawNumberToAddressConverter);
46             }
47         }
48
49         private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
        ↪ meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
        ↪ numberToAddressConverter)
50         {
51             var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
52             var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
        ↪ addressToNumberConverter, unicodeSymbolMarker);
53             var originalCharacter = 'H';
54             var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);

```

```

55     var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
    ↪ unicodeSymbolMarker);
56     var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
    ↪ numberToAddressConverter, unicodeSymbolCriterionMatcher);
57     var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
58     Assert.Equal(originalCharacter, resultingCharacter);
59 }
60
61 [Fact]
62 public static void StringAndUnicodeSequenceConvertersTest()
63 {
64     using (var scope = new TempLinksTestScope())
65     {
66         var links = scope.Links;
67
68         var itself = links.Constants.Itself;
69
70         var meaningRoot = links.CreatePoint();
71         var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
72         var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
73         var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
74         var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
75         var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
76
77         var powerOf2ToUnaryNumberConverter = new
    ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
78         var addressToUnaryNumberConverter = new
    ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
79         var charToUnicodeSymbolConverter = new
    ↪ CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
    ↪ unicodeSymbolMarker);
80
81         var unaryNumberToAddressConverter = new
    ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
    ↪ powerOf2ToUnaryNumberConverter);
82         var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
83         var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
    ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
84         var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
    ↪ frequencyPropertyMarker, frequencyMarker);
85         var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
    ↪ frequencyPropertyOperator, frequencyIncrementer);
86         var linkToItsFrequencyNumberConverter = new
    ↪ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
    ↪ unaryNumberToAddressConverter);
87         var sequenceToItsLocalElementLevelsConverter = new
    ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
    ↪ linkToItsFrequencyNumberConverter);
88         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
    ↪ sequenceToItsLocalElementLevelsConverter);
89
90         var stringToUnicodeSequenceConverter = new
    ↪ StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
    ↪ index, optimalVariantConverter, unicodeSequenceMarker);
91
92         var originalString = "Hello";
93
94         var unicodeSequenceLink =
    ↪ stringToUnicodeSequenceConverter.Convert(originalString);
95
96         var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
    ↪ unicodeSymbolMarker);
97         var unicodeSymbolToCharConverter = new
    ↪ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
    ↪ unicodeSymbolCriterionMatcher);
98
99         var unicodeSequenceCriterionMatcher = new TargetMatcher<ulong>(links,
    ↪ unicodeSequenceMarker);
100
101         var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
    ↪ unicodeSymbolCriterionMatcher.IsMatched);
102
103         var unicodeSequenceToStringConverter = new
    ↪ UnicodeSequenceToStringConverter<ulong>(links,
    ↪ unicodeSequenceCriterionMatcher, sequenceWalker,
    ↪ unicodeSymbolToCharConverter);
104

```

```

105         var resultingString =
106             ↪ unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
107         Assert.Equal(originalString, resultingString);
108     }
109 }
110 }
111 }

```

1.195 ./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt32LinksTests.cs

```

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

```

1.196 ./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().TestMultip
            ↳ leRandomCreationsAndDeletions(100));
29     }
30
31     private static void Using(Action<ILinks<TLink>> action)
32     {
33         using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↳ UInt64UnitedMemoryLinks>>())
34         {
35             action(scope.Use<ILinks<TLink>>());
36         }
37     }
38 }
39 }

```

Index

./csharp/Platform.Data.Doublets.Tests/BigIntegerConvertersTests.cs, 238
./csharp/Platform.Data.Doublets.Tests/DefaultSequenceAppenderTests.cs, 240
./csharp/Platform.Data.Doublets.Tests/GenericLinksTests.cs, 241
./csharp/Platform.Data.Doublets.Tests/ILinksExtensionsTests.cs, 241
./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs, 242
./csharp/Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs, 242
./csharp/Platform.Data.Doublets.Tests/RationalNumbersTests.cs, 245
./csharp/Platform.Data.Doublets.Tests/ReadSequenceTests.cs, 247
./csharp/Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs, 248
./csharp/Platform.Data.Doublets.Tests/ScopeTests.cs, 249
./csharp/Platform.Data.Doublets.Tests/SequencesTests.cs, 250
./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs, 264
./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt32LinksTests.cs, 265
./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt64LinksTests.cs, 266
./csharp/Platform.Data.Doublets.Tests/TempLinksTestScope.cs, 267
./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs, 267
./csharp/Platform.Data.Doublets.Tests/UInt64LinksTests.cs, 270
./csharp/Platform.Data.Doublets.Tests/UInt64LinksExtensionsTests.cs, 282
./csharp/Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs, 283
./csharp/Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs, 284
./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt32LinksTests.cs, 286
./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt64LinksTests.cs, 286
./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/Incrementers/FrequencyIncrementer.cs, 28
./csharp/Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs, 28
./csharp/Platform.Data.Doublets/Link.cs, 29
./csharp/Platform.Data.Doublets/LinkExtensions.cs, 32
./csharp/Platform.Data.Doublets/LinksOperatorBase.cs, 32
./csharp/Platform.Data.Doublets/Memory/ILinksListMethods.cs, 32
./csharp/Platform.Data.Doublets/Memory/ILinksTreeMethods.cs, 33
./csharp/Platform.Data.Doublets/Memory/IndexTreeType.cs, 33
./csharp/Platform.Data.Doublets/Memory/LinksHeader.cs, 33
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksRecursionlessSizeBalancedTreeMethodsBase.cs, 34
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSizeBalancedTreeMethodsBase.cs, 37
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 40
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSourcesSizeBalancedTreeMethods.cs, 41
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 42
./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsSizeBalancedTreeMethods.cs, 43
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksRecursionlessSizeBalancedTreeMethodsBase.cs, 44
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSizeBalancedTreeMethodsBase.cs, 46
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesLinkedListMethods.cs, 49
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesRecursionlessSizeBalancedTreeMethods.cs, 50
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesSizeBalancedTreeMethods.cs, 51
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 52
./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsSizeBalancedTreeMethods.cs, 53
./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinks.cs, 54

[illegible]

./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs, 141
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsRecursionlessSizeBalancedTreeMethods.cs, 142
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs, 143
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnitedMemoryLinks.cs, 144
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnusedLinksListMethods.cs, 146
./csharp/Platform.Data.Doublets/Numbers/Rational/DecimalToRationalConverter.cs, 146
./csharp/Platform.Data.Doublets/Numbers/Rational/RationalToDecimalConverter.cs, 147
./csharp/Platform.Data.Doublets/Numbers/Raw/BigIntegerToRawNumberSequenceConverter.cs, 148
./csharp/Platform.Data.Doublets/Numbers/Raw/LongRawNumberSequenceToNumberConverter.cs, 148
./csharp/Platform.Data.Doublets/Numbers/Raw/NumberToLongRawNumberSequenceConverter.cs, 149
./csharp/Platform.Data.Doublets/Numbers/Raw/RawNumberSequenceToBigIntegerConverter.cs, 150
./csharp/Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs, 150
./csharp/Platform.Data.Doublets/Numbers/Unary/LinkTotlsFrequencyNumberConveter.cs, 151
./csharp/Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs, 152
./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs, 152
./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs, 153
./csharp/Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 154
./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 155
./csharp/Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 156
./csharp/Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 157
./csharp/Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 160
./csharp/Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 160
./csharp/Platform.Data.Doublets/Sequences/Converters/SequenceTotlsLocalElementLevelsConverter.cs, 162
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs, 162
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs, 163
./csharp/Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 163
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 164
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 164
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 167
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 168
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkTotlsFrequencyValueConverter.cs, 169
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 169
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 169
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 170
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 171
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 171
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 171
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 172
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 173
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 173
./csharp/Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 173
./csharp/Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 174
./csharp/Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 175
./csharp/Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 175
./csharp/Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 176
./csharp/Platform.Data.Doublets/Sequences/Indexes/Unindex.cs, 177
./csharp/Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 177
./csharp/Platform.Data.Doublets/Sequences/Sequences.cs, 204
./csharp/Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 215
./csharp/Platform.Data.Doublets/Sequences/SequencesOptions.cs, 215
./csharp/Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 218
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 218
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 219
./csharp/Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 221
./csharp/Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 221
./csharp/Platform.Data.Doublets/Stacks/Stack.cs, 222
./csharp/Platform.Data.Doublets/Stacks/StackExtensions.cs, 223
./csharp/Platform.Data.Doublets/SynchronizedLinks.cs, 223
./csharp/Platform.Data.Doublets/Time/DateTimeToLongRawNumberSequenceConverter.cs, 224
./csharp/Platform.Data.Doublets/Time/LongRawNumberSequenceToDateTimeConverter.cs, 224
./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs, 225
./csharp/Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 227
./csharp/Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 232
./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 233
./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSymbolsListConverter.cs, 234
./csharp/Platform.Data.Doublets/Unicode/UnicodeMap.cs, 234
./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 237

./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 237
./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.cs, 238