

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

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

```

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

```

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

```

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

```

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

```

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

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

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

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

1.14 ./csharp/Platform.Data.Doublets/Decorators/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)
```

```

19  /// + меньше объём БД
20  /// - больше сложность
21  ///
22  /// Текущее теоретическое ограничение на индекс связи, из-за использования 5 бит в размере
  ↳ поддеревьев для AVL баланса и флагов нитей: 2 в степени (64 минус 5 равно 59 ) равно 576
  ↳ 460 752 303 423 488
23  /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
  ↳ (битовыми строками) - вариант матрицы (выстраиваемой лениво).
24  ///
25  /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
  ↳ выбрасываться только при #if DEBUG
26  /// </remarks>
27  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] !=
  ↳ newTarget)
57              {
58                  links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
  ↳ newSource,
59                                  newTarget == itselfConstant ? updatedLink :
  ↳ newTarget);
60              }
61              return updatedLink;
62          }
63          else
64          {
65              return _facade.MergeAndDelete(updatedLink, existedLink);
66          }
67      }
68
69      [MethodImpl(MethodImplOptions.AggressiveInlining)]
70      public override void Delete(IList<ulong> restrictions)
71      {
72          var linkIndex = restrictions[_constants.IndexPart];
73          var links = _links;
74          links.EnforceResetValues(linkIndex);
75          _facade.DeleteAllUsages(linkIndex);
76          links.Delete(linkIndex);
77      }
78  }
79  }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using Platform.Collections;
5  using Platform.Collections.Lists;
6  using Platform.Data.Universal;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9

```

```

10 namespace Platform.Data.Doublets.Decorators
11 {
12     /// <remarks>
13     /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
14     /// Now we go with nothing. And nothing is something one, but empty, and cannot be changed
15     /// ↪ by itself. But can cause creation (update from nothing) or deletion (update to nothing).
16     ///
17     /// TODO: Decide to change to IDoubletLinks or not to change. (Better to create
18     /// ↪ DefaultUniLinksBase, that contains logic itself and can be implemented using both
19     /// ↪ IDoubletLinks and ILinks.)
20     /// </remarks>
21     internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
22     {
23         private static readonly EqualityComparer<TLink> _equalityComparer =
24             ↪ EqualityComparer<TLink>.Default;
25
26         public UniLinks(ILinks<TLink> links) : base(links) { }
27
28         private struct Transition
29         {
30             public IList<TLink> Before;
31             public IList<TLink> After;
32
33             public Transition(IList<TLink> before, IList<TLink> after)
34             {
35                 Before = before;
36                 After = after;
37             }
38         }
39
40         //public static readonly TLink NullConstant = Use<LinksConstants<TLink>>.Single.Null;
41         //public static readonly IReadOnlyList<TLink> NullLink = new
42         ↪ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
43         ↪ });
44
45         // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
46         ↪ (Links-Expression)
47         public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
48         ↪ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
49         ↪ substitutedHandler)
50         {
51             ///List<Transition> transitions = null;
52             ///if (!restriction.IsNullOrEmpty())
53             ///{
54             ///    // Есть причина делать проход (чтение)
55             ///    if (matchedHandler != null)
56             ///    {
57             ///        if (!substitution.IsNullOrEmpty())
58             ///        {
59             ///            // restriction => { 0, 0, 0 } | { 0 } // Create
60             ///            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
61             ↪ Create / Update
62             ///            // substitution => { 0, 0, 0 } | { 0 } // Delete
63             ///            transitions = new List<Transition>();
64             ///            if (Equals(substitution[Constants.IndexPart], Constants.Null))
65             ///            {
66             ///                // If index is Null, that means we always ignore every other
67             ↪ value (they are also Null by definition)
68             ///                var matchDecision = matchedHandler(, NullLink);
69             ///                if (Equals(matchDecision, Constants.Break))
70             ///                    return false;
71             ///                if (!Equals(matchDecision, Constants.Skip))
72             ///                    transitions.Add(new Transition(matchedLink, newValue));
73             ///            }
74             ///            else
75             ///            {
76             ///                Func<T, bool> handler;
77             ///                handler = link =>
78             ///                {
79             ///                    var matchedLink = Memory.GetLinkValue(link);
80             ///                    var newValue = Memory.GetLinkValue(link);
81             ///                    newValue[Constants.IndexPart] = Constants.Itself;
82             ///                    newValue[Constants.SourcePart] =
83             ↪ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
84             ↪ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
85             ///                    newValue[Constants.TargetPart] =
86             ↪ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
87             ↪ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];

```



```

73         var matchDecision = matchedHandler(matchedLink, newValue);
74         if (Equals(matchDecision, Constants.Break))
75             return false;
76         if (!Equals(matchDecision, Constants.Skip))
77             transitions.Add(new Transition(matchedLink, newValue));
78         return true;
79     };
80     if (!Memory.Each(handler, restriction))
81         return Constants.Break;
82     }
83     }
84     else
85     {
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 //    {

```

```

150         //         matchedLinks = new List<ILink<T>>>();
151         //         Func<T, bool> handler = link =>
152         //         {
153         //             var matchedLink = Memory.GetLinkValue(link);
154         //             var matchDecision = matchedHandler(matchedLink);
155         //             if (Equals(matchDecision, Constants.Break))
156         //                 return false;
157         //             if (!Equals(matchDecision, Constants.Skip))
158         //                 matchedLinks.Add(matchedLink);
159         //             return true;
160         //         };
161         //         if (!Memory.Each(handler, restriction))
162         //             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         return _constants.Continue;
194     }
195
196     public TLink Trigger(ILink<TLink> patternOrCondition, Func<ILink<TLink>, TLink>
197     ↪ matchHandler, ILink<TLink> substitution, Func<ILink<TLink>, ILink<TLink>, TLink>
198     ↪ substitutionHandler)
199     {
200         var constants = _constants;
201         if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
202         {
203             return constants.Continue;
204         }
205         else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
206         ↪ Check if it is a correct condition
207         {
208             // Or it only applies to trigger without matchHandler.
209             throw new NotImplementedException();
210         }
211         else if (!substitution.IsNullOrEmpty()) // Creation
212         {
213             var before = Array.Empty<TLink>();
214             // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
215             ↪ (пройти мимо) или пустить (взять)?
216             if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
217             ↪ constants.Break))
218             {
219                 return constants.Break;
220             }
221             var after = (ILink<TLink>)substitution.ToArray();
222             if (_equalityComparer.Equals(after[0], default))
223             {
224                 var newLink = _links.Create();
225                 after[0] = newLink;
226             }
227             if (substitution.Count == 1)

```

```

220     {
221         after = _links.GetLink(substitution[0]);
222     }
223     else if (substitution.Count == 3)
224     {
225         //Links.Create(after);
226     }
227     else
228     {
229         throw new NotSupportedException();
230     }
231     if (matchHandler != null)
232     {
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     }
300 }

```

```

    }
    return constants.Continue;
}
else
{
    throw new NotSupportedException();
}
}

}

/// <remarks>
/// IList[IList[IList[T]]]
/// |           |           |
/// |           |-----|
/// |           |   link   |
/// |-----|
/// |       change         |
/// |-----|
/// |       changes        |
/// </remarks>
public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink>
↪ substitution)
{
    var changes = new List<IList<IList<TLink>>>>();
    var @continue = _constants.Continue;
    Trigger(condition, AlwaysContinue, substitution, (before, after) =>
    {
        var change = new[] { before, after };
        changes.Add(change);
        return @continue;
    });
    return changes;
}

private TLink AlwaysContinue(IList<TLink> linkToMatch) => _constants.Continue;

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets
8 {
9     public struct Doublet<T> : IEquatable<Doublet<T>>
10     {
11         private static readonly EqualityComparer<T> _equalityComparer =
12             ↳ EqualityComparer<T>.Default;
13
14         public readonly T Source;
15
16         public readonly T Target;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public Doublet(T source, T target)
20         {
21             Source = source;
22             Target = target;
23         }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public override string ToString() => $"{Source}->{Target}";
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
30             ↳ && _equalityComparer.Equals(Target, other.Target);
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public override bool Equals(object obj) => obj is Doublet<T> doublet ?
34             ↳ base.Equals(doublet) : false;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public override int GetHashCode() => (Source, Target).GetHashCode();
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public static bool operator ==(Doublet<T> left, Doublet<T> right) => left.Equals(right);
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         public static bool operator !=(Doublet<T> left, Doublet<T> right) => !left.Equals(right);
44     }
45 }

```

```

39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public static bool operator !=(Doublet<T> left, Doublet<T> right) => !(left == right);
41     }
42 }

```

1.17 ./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 }

```

1.18 ./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 }

```

1.19 ./csharp/Platform.Data.Doublets/ILinksExtensions.cs

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Runtime.CompilerServices;
6  using Platform.Ranges;
7  using Platform.Collections.Arrays;
8  using Platform.Random;
9  using Platform.Setters;
10 using Platform.Converters;
11 using Platform.Numbers;
12 using Platform.Data.Exceptions;
13 using Platform.Data.Doublets.Decorators;
14
15 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
16
17 namespace Platform.Data.Doublets
18 {
19     public static class ILinksExtensions
20     {
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public static void RunRandomCreations<TLink>(this ILinks<TLink> links, ulong
23             ↳ amountOfCreations)
24         {
25             var random = RandomHelpers.Default;
26             var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
27             var uint64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
28             for (var i = 0UL; i < amountOfCreations; i++)
29             {
30                 var linksAddressRange = new Range<ulong>(0,
31                     ↳ addressToUInt64Converter.Convert(links.Count()));
32                 var source =
33                     ↳ uint64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
34                 var target =
35                     ↳ uint64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
36                 links.GetOrCreate(source, target);
37             }
38         }
39     }
40 }

```

```

35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 public static void RunRandomSearches<TLink>(this ILinks<TLink> links, ulong
37     ↳ amountOfSearches)
38 {
39     var random = RandomHelpers.Default;
40     var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
41     var uint64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
42     for (var i = 0UL; i < amountOfSearches; i++)
43     {
44         var linksAddressRange = new Range<ulong>(0,
45             ↳ addressToUInt64Converter.Convert(links.Count()));
46         var source =
47             ↳ uint64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
48         var target =
49             ↳ uint64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
50         links.SearchOrDefault(source, target);
51     }
52 }
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, ulong
56     ↳ amountOfDeletions)
57 {
58     var random = RandomHelpers.Default;
59     var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
60     var uint64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
61     var linksCount = addressToUInt64Converter.Convert(links.Count());
62     var min = amountOfDeletions > linksCount ? 0UL : linksCount - amountOfDeletions;
63     for (var i = 0UL; i < amountOfDeletions; i++)
64     {
65         linksCount = addressToUInt64Converter.Convert(links.Count());
66         if (linksCount <= min)
67         {
68             break;
69         }
70         var linksAddressRange = new Range<ulong>(min, linksCount);
71         var link =
72             ↳ uint64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
73         links.Delete(link);
74     }
75 }
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 public static void Delete<TLink>(this ILinks<TLink> links, TLink linkToDelete) =>
79     ↳ links.Delete(new LinkAddress<TLink>(linkToDelete));
80
81 /// <remarks>
82 /// TODO: Возможно есть очень простой способ это сделать.
83 /// (Например просто удалить файл, или изменить его размер таким образом,
84 /// чтобы удалится весь контент)
85 /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
86 /// </remarks>
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 public static void DeleteAll<TLink>(this ILinks<TLink> links)
89 {
90     var equalityComparer = EqualityComparer<TLink>.Default;
91     var comparer = Comparer<TLink>.Default;
92     for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
93         ↳ Arithmetic.Decrement(i))
94     {
95         links.Delete(i);
96         if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
97         {
98             i = links.Count();
99         }
100     }
101 }
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 public static TLink First<TLink>(this ILinks<TLink> links)
105 {
106     TLink firstLink = default;
107     var equalityComparer = EqualityComparer<TLink>.Default;
108     if (equalityComparer.Equals(links.Count(), default))
109     {
110         throw new InvalidOperationException("В хранилище нет связей.");
111     }
112 }

```

```

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

```

```

177         //throw new InvalidOperationException(string.Format("Невозможно продолжить
178         ↪ путь через элемент пути {0}", next));
179         return false;
180     }
181     current = next;
182 }
183 return true;
184 }
185 /// <remarks>
186 /// Может потребовать дополнительного стека для PathElement's при использовании
187 ↪ SequenceWalker.
188 /// </remarks>
189 [MethodImpl(MethodImplOptions.AggressiveInlining)]
190 public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink root, params int[]
191 ↪ path)
192 {
193     links.EnsureLinkExists(root, "root");
194     var currentLink = root;
195     for (var i = 0; i < path.Length; i++)
196     {
197         currentLink = links.GetLink(currentLink)[path[i]];
198     }
199     return currentLink;
200 }
201 [MethodImpl(MethodImplOptions.AggressiveInlining)]
202 public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
203 ↪ links, TLink root, ulong size, ulong index)
204 {
205     var constants = links.Constants;
206     var source = constants.SourcePart;
207     var target = constants.TargetPart;
208     if (!Platform.Numbers.Math.IsPowerOfTwo(size))
209     {
210         throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
211         ↪ than powers of two are not supported.");
212     }
213     var path = new BitArray(BitConverter.GetBytes(index));
214     var length = Bit.GetLowestPosition(size);
215     links.EnsureLinkExists(root, "root");
216     var currentLink = root;
217     for (var i = length - 1; i >= 0; i--)
218     {
219         currentLink = links.GetLink(currentLink)[path[i] ? target : source];
220     }
221     return currentLink;
222 }
223 #endregion
224 /// <summary>
225 /// Возвращает индекс указанной связи.
226 /// </summary>
227 /// <param name="links">Хранилище связей.</param>
228 /// <param name="link">Связь представленная списком, состоящим из её адреса и
229 ↪ содержимого.</param>
230 /// <returns>Индекс начальной связи для указанной связи.</returns>
231 [MethodImpl(MethodImplOptions.AggressiveInlining)]
232 public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
233 ↪ link[links.Constants.IndexPart];
234
235 /// <summary>
236 /// Возвращает индекс начальной (Source) связи для указанной связи.
237 /// </summary>
238 /// <param name="links">Хранилище связей.</param>
239 /// <param name="link">Индекс связи.</param>
240 /// <returns>Индекс начальной связи для указанной связи.</returns>
241 [MethodImpl(MethodImplOptions.AggressiveInlining)]
242 public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
243 ↪ links.GetLink(link)[links.Constants.SourcePart];
244
245 /// <summary>
246 /// Возвращает индекс начальной (Source) связи для указанной связи.
247 /// </summary>
248 /// <param name="links">Хранилище связей.</param>
249 /// <param name="link">Связь представленная списком, состоящим из её адреса и
250 ↪ 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) =>
249     ↪ link[links.Constants.SourcePart];
250
251 /// <summary>
252 /// Возвращает индекс конечной (Target) связи для указанной связи.
253 /// </summary>
254 /// <param name="links">Хранилище связей.</param>
255 /// <param name="link">Индекс связи.</param>
256 /// <returns>Индекс конечной связи для указанной связи.</returns>
257 [MethodImpl(MethodImplOptions.AggressiveInlining)]
258 public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
259     ↪ links.GetLink(link)[links.Constants.TargetPart];
260
261 /// <summary>
262 /// Возвращает индекс конечной (Target) связи для указанной связи.
263 /// </summary>
264 /// <param name="links">Хранилище связей.</param>
265 /// <param name="link">Связь представленная списком, состоящим из её адреса и
266     ↪ содержимого.</param>
267 /// <returns>Индекс конечной связи для указанной связи.</returns>
268 [MethodImpl(MethodImplOptions.AggressiveInlining)]
269 public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
270     ↪ link[links.Constants.TargetPart];
271
272 /// <summary>
273 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
274     ↪ (handler) для каждой подходящей связи.
275 /// </summary>
276 /// <param name="links">Хранилище связей.</param>
277 /// <param name="handler">Обработчик каждой подходящей связи.</param>
278 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
279     ↪ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
280     ↪ Any - отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
281 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
282     ↪ случае.</returns>
283 [MethodImpl(MethodImplOptions.AggressiveInlining)]
284 public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
285     ↪ handler, params TLink[] restrictions)
286     => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
287         ↪ links.Constants.Continue);
288
289 /// <summary>
290 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
291     ↪ (handler) для каждой подходящей связи.
292 /// </summary>
293 /// <param name="links">Хранилище связей.</param>
294 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
295     ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
296     ↪ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
297 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
298     ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
299     ↪ 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
302     ↪ случае.</returns>
303 [MethodImpl(MethodImplOptions.AggressiveInlining)]
304 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
305     ↪ Func<TLink, bool> handler)
306 {
307     var constants = links.Constants;
308     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
309         ↪ constants.Break, constants.Any, source, target);
310 }
311
312 /// <summary>
313 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
314     ↪ (handler) для каждой подходящей связи.
315 /// </summary>
316 /// <param name="links">Хранилище связей.</param>
317 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
318     ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
319     ↪ Constants.Any - любое начало, 1..∞ конкретное начало)</param>

```

```

399  /// <param name="target">Значение, определяющее соответствующие шаблону связи.
400  ↪ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
401  ↪ 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
402  ↪ случае.</returns>
302  [MethodImpl(MethodImplOptions.AggressiveInlining)]
303  public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
404  ↪ Func<IList<TLink>, TLink> handler) => links.Each(handler, links.Constants.Any,
405  ↪ source, target);
304
305  [MethodImpl(MethodImplOptions.AggressiveInlining)]
306  public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
407  ↪ restrictions)
307  {
308      var arraySize = CheckedConverter<TLink,
409  ↪ 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,
410  ↪ 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[]
411  ↪ restrictions)
324  {
325      var arraySize = CheckedConverter<TLink,
412  ↪ 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
413  ↪ в хранилище связей.
341  /// </summary>
342  /// <param name="links">Хранилище связей.</param>
343  /// <param name="source">Начало связи.</param>
344  /// <param name="target">Конец связи.</param>
345  /// <returns>Значение, определяющее существует ли связь.</returns>
346  [MethodImpl(MethodImplOptions.AggressiveInlining)]
347  public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
414  ↪ => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
415  ↪ 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>
416  ↪ 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],
417  ↪ $"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 ArgumentException<TLink>(link);
433     }
434 }
435
436 /// <param name="links">Хранилище связей.</param>
437 [MethodImpl(MethodImplOptions.AggressiveInlining)]
438 public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ addresses) => links.EnsureCreated(links.Create, addresses);
439
440 /// <param name="links">Хранилище связей.</param>
441 [MethodImpl(MethodImplOptions.AggressiveInlining)]
442 public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
443
444 /// <param name="links">Хранилище связей.</param>
445 [MethodImpl(MethodImplOptions.AggressiveInlining)]
446 public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
    ↳ params TLink[] addresses)
447 {
448     var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
449     var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
450     var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
    ↳ !links.Exists(x)));
451     if (nonExistentAddresses.Count > 0)
452     {
453         var max = nonExistentAddresses.Max();
454         max = uInt64ToAddressConverter.Convert(System.Math.Min(addressToUInt64Converter.
    ↳ Convert(max),
    ↳ addressToUInt64Converter.Convert(links.Constants.InternalReferencesRange.Max
    ↳ imum)));
455         var createdLinks = new List<TLink>();
456         var equalityComparer = EqualityComparer<TLink>.Default;
457         TLink createdLink = creator();
458         while (!equalityComparer.Equals(createdLink, max))
459         {
460             createdLinks.Add(createdLink);
461         }
462         for (var i = 0; i < createdLinks.Count; i++)
463         {
464             if (!nonExistentAddresses.Contains(createdLinks[i]))
465             {
466                 links.Delete(createdLinks[i]);
467             }
468         }
469     }
470 }
471
472 #endregion
473
474 /// <param name="links">Хранилище связей.</param>
475 [MethodImpl(MethodImplOptions.AggressiveInlining)]
476 public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
477 {
478     var constants = links.Constants;
479     var values = links.GetLink(link);
480     TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link,
    ↳ constants.Any));
481     var equalityComparer = EqualityComparer<TLink>.Default;
482     if (equalityComparer.Equals(values[constants.SourcePart], link))
483     {
484         usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
485     }
486     TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
    ↳ link));
487     if (equalityComparer.Equals(values[constants.TargetPart], link))
488     {
489         usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
490     }
491     return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
492 }
493
494 /// <param name="links">Хранилище связей.</param>
495 [MethodImpl(MethodImplOptions.AggressiveInlining)]
496 public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
    ↳ Comparer<TLink>.Default.Compare(links.CountUsages(link), default) > 0;
497
498 /// <param name="links">Хранилище связей.</param>

```

```

499 [MethodImpl(MethodImplOptions.AggressiveInlining)]
500 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
    ↪ TLink target)
501 {
502     var constants = links.Constants;
503     var values = links.GetLink(link);
504     var equalityComparer = EqualityComparer<TLink>.Default;
505     return equalityComparer.Equals(values[constants.SourcePart], source) &&
    ↪ equalityComparer.Equals(values[constants.TargetPart], target);
506 }
507
508 /// <summary>
509 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
510 /// </summary>
511 /// <param name="links">Хранилище связей.</param>
512 /// <param name="source">Индекс связи, которая является началом для искомой
    ↪ связи.</param>
513 /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
514 /// <returns>Индекс искомой связи с указанными Source (началом) и Target
    ↪ (концом).</returns>
515 [MethodImpl(MethodImplOptions.AggressiveInlining)]
516 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↪ target)
517 {
518     var constants = links.Constants;
519     var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break, default);
520     links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
521     return setter.Result;
522 }
523
524 /// <param name="links">Хранилище связей.</param>
525 [MethodImpl(MethodImplOptions.AggressiveInlining)]
526 public static TLink Create<TLink>(this ILinks<TLink> links) => links.Create(null);
527
528 /// <param name="links">Хранилище связей.</param>
529 [MethodImpl(MethodImplOptions.AggressiveInlining)]
530 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
531 {
532     var link = links.Create();
533     return links.Update(link, link, link);
534 }
535
536 /// <param name="links">Хранилище связей.</param>
537 [MethodImpl(MethodImplOptions.AggressiveInlining)]
538 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↪ target) => links.Update(links.Create(), source, target);
539
540 /// <summary>
541 /// Обновляет связь с указанными началом (Source) и концом (Target)
542 /// на связь с указанными началом (NewSource) и концом (NewTarget).
543 /// </summary>
544 /// <param name="links">Хранилище связей.</param>
545 /// <param name="link">Индекс обновляемой связи.</param>
546 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    ↪ 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],
569             ↪ restrictions[2], restrictions[3]);
570     }
571     else
572     {
573         return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
574     }
575 }
576 [MethodImpl(MethodImplOptions.AggressiveInlining)]
577 public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
578     ↪ links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
579 {
580     var equalityComparer = EqualityComparer<TLink>.Default;
581     var constants = links.Constants;
582     var restrictionsIndex = restrictions[constants.IndexPart];
583     var substitutionIndex = substitution[constants.IndexPart];
584     if (equalityComparer.Equals(substitutionIndex, default))
585     {
586         substitutionIndex = restrictionsIndex;
587     }
588     var source = substitution[constants.SourcePart];
589     var target = substitution[constants.TargetPart];
590     source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
591     target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
592     return new Link<TLink>(substitutionIndex, source, target);
593 }
594 /// <summary>
595 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
596 ↪ с указанными Source (началом) и Target (концом).
597 /// </summary>
598 /// <param name="links">Хранилище связей.</param>
599 /// <param name="source">Индекс связи, которая является началом на создаваемой
600 ↪ связи.</param>
601 /// <param name="target">Индекс связи, которая является концом для создаваемой
602 ↪ связи.</param>
603 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
604 [MethodImpl(MethodImplOptions.AggressiveInlining)]
605 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
606     ↪ target)
607 {
608     var link = links.SearchOrDefault(source, target);
609     if (EqualityComparer<TLink>.Default.Equals(link, default))
610     {
611         link = links.CreateAndUpdate(source, target);
612     }
613     return link;
614 }
615 /// <summary>
616 /// Обновляет связь с указанными началом (Source) и концом (Target)
617 /// на связь с указанными началом (NewSource) и концом (NewTarget).
618 /// </summary>
619 /// <param name="links">Хранилище связей.</param>
620 /// <param name="source">Индекс связи, которая является началом обновляемой
621 ↪ связи.</param>
622 /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
623 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
624 ↪ выполняется обновление.</param>
625 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
626 ↪ выполняется обновление.</param>
627 /// <returns>Индекс обновлённой связи.</returns>
628 [MethodImpl(MethodImplOptions.AggressiveInlining)]
629 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
630     ↪ TLink target, TLink newSource, TLink newTarget)
631 {
632     var equalityComparer = EqualityComparer<TLink>.Default;
633     var link = links.SearchOrDefault(source, target);
634     if (equalityComparer.Equals(link, default))
635     {
636         return links.CreateAndUpdate(newSource, newTarget);
637     }
638 }

```

```

631         if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
632             ↪ target))
633         {
634             return link;
635         }
636         return links.Update(link, newSource, newTarget);
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
644         ↪ target)
645     {
646         var link = links.SearchOrDefault(source, target);
647         if (!EqualityComparer<TLink>.Default.Equals(link, default))
648         {
649             links.Delete(link);
650             return link;
651         }
652         return default;
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     /// <remarks>Before execution of this method ensure that deleted link is detached (all
666     ↪ values - source and target are reset to null) or it might enter into infinite
667     ↪ recursion.</remarks>
668     [MethodImpl(MethodImplOptions.AggressiveInlining)]
669     public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
670     {
671         var anyConstant = links.Constants.Any;
672         var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
673         links.DeleteByQuery(usagesAsSourceQuery);
674         var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
675         links.DeleteByQuery(usagesAsTargetQuery);
676     }
677     [MethodImpl(MethodImplOptions.AggressiveInlining)]
678     public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
679     {
680         var count = CheckedConverter<TLink, long>.Default.Convert(links.Count(query));
681         if (count > 0)
682         {
683             var queryResult = new TLink[count];
684             var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
685                 ↪ links.Constants.Continue);
686             links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
687             for (var i = count - 1; i >= 0; i--)
688             {
689                 links.Delete(queryResult[i]);
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));
752         var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
753 → usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
754         if (!isStandalonePoint)
755         {
756             var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
757             if (totalUsages > 0)
758             {
759                 var usages = ArrayPool.Allocate<TLink>(totalUsages);
760                 var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
761 → links.Constants.Continue);
762                 var i = 0L;
763                 if (usagesAsSourceCount > 0)
764                 {
765                     links.Each(usagesFiller.AddFirstAndReturnConstant,
766 → usagesAsSourceQuery);
767                     for (; i < usagesAsSourceCount; i++)
768                     {
769                         var usage = usages[i];
770                         if (!equalityComparer.Equals(usage, oldLinkIndex))
771                         {
772                             links.Update(usage, newLinkIndex, links.GetTarget(usage));
773                         }
774                     }
775                 }
776                 if (usagesAsTargetCount > 0)
777                 {
778                     links.Each(usagesFiller.AddFirstAndReturnConstant,
779 → usagesAsTargetQuery);
780                     for (; i < usages.Length; i++)
781                     {

```



```

770         var usage = usages[i];
771         if (!equalityComparer.Equals(usage, oldLinkIndex))
772         {
773             links.Update(usage, links.GetSource(usage), newLinkIndex);
774         }
775     }
776 }
777 ArrayPool.Free(usages);
778 }
779 }
780 }
781 return newLinkIndex;
782 }
783
784 /// <summary>
785 /// Replace one link with another (replaced link is deleted, children are updated or
786   ↳ deleted).
787 /// </summary>
788 [MethodImpl(MethodImplOptions.AggressiveInlining)]
789 public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
790   ↳ TLink newLinkIndex)
791 {
792     var equalityComparer = EqualityComparer<TLink>.Default;
793     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
794     {
795         links.MergeUsages(oldLinkIndex, newLinkIndex);
796         links.Delete(oldLinkIndex);
797     }
798     return newLinkIndex;
799 }
800
801 [MethodImpl(MethodImplOptions.AggressiveInlining)]
802 public static ILinks<TLink>
803   ↳ DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
804 {
805     links = new LinksCascadeUsagesResolver<TLink>(links);
806     links = new NonNullContentsLinkDeletionResolver<TLink>(links);
807     links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
808     return links;
809 }
810
811 [MethodImpl(MethodImplOptions.AggressiveInlining)]
812 public static string Format<TLink>(this ILinks<TLink> links, IList<TLink> link)
813 {
814     var constants = links.Constants;
815     return $"({link[constants.IndexPart]}: {link[constants.SourcePart]})
816   ↳ {link[constants.TargetPart]}";
817 }
818
819 [MethodImpl(MethodImplOptions.AggressiveInlining)]
820 public static string Format<TLink>(this ILinks<TLink> links, TLink link) =>
821   ↳ links.Format(links.GetLink(link));
822 }
823 }

```

1.20 ./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  }

```

1.21 ./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;

```

```

12
13     private readonly TLink _frequencyMarker;
14     private readonly TLink _unaryOne;
15     private readonly IIncrementer<TLink> _unaryNumberIncrementer;
16
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
19         ↳ IIncrementer<TLink> unaryNumberIncrementer)
20         : base(links)
21     {
22         _frequencyMarker = frequencyMarker;
23         _unaryOne = unaryOne;
24         _unaryNumberIncrementer = unaryNumberIncrementer;
25     }
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     public TLink Increment(TLink frequency)
29     {
30         var links = _links;
31         if (_equalityComparer.Equals(frequency, default))
32         {
33             return links.GetOrCreate(_unaryOne, _frequencyMarker);
34         }
35         var incrementedSource =
36             ↳ _unaryNumberIncrementer.Increment(links.GetSource(frequency));
37         return links.GetOrCreate(incrementedSource, _frequencyMarker);
38     }

```

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

```

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

```

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

```

1  using Platform.Collections.Lists;
2  using Platform.Exceptions;
3  using Platform.Ranges;
4  using Platform.Singletons;
5  using System;
6  using System.Collections;
7  using System.Collections.Generic;

```

```

8 using System.Runtime.CompilerServices;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets
13 {
14     /// <summary>
15     /// Структура описывающая уникальную связь.
16     /// </summary>
17     public struct Link<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>, IList<TLink>
18     {
19         public static readonly Link<TLink> Null = new Link<TLink>();
20
21         private static readonly LinksConstants<TLink> _constants =
22             ↪ Default<LinksConstants<TLink>>.Instance;
23         private static readonly EqualityComparer<TLink> _equalityComparer =
24             ↪ EqualityComparer<TLink>.Default;
25
26         private const int Length = 3;
27
28         public readonly TLink Index;
29         public readonly TLink Source;
30         public readonly TLink Target;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public Link(params TLink[] values) => SetValues(values, out Index, out Source, out
34             ↪ Target);
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public Link(IList<TLink> values) => SetValues(values, out Index, out Source, out Target);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public Link(object other)
41         {
42             if (other is Link<TLink> otherLink)
43             {
44                 SetValues(ref otherLink, out Index, out Source, out Target);
45             }
46             else if (other is IList<TLink> otherList)
47             {
48                 SetValues(otherList, out Index, out Source, out Target);
49             }
50             else
51             {
52                 throw new NotSupportedException();
53             }
54         }
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         public Link(ref Link<TLink> other) => SetValues(ref other, out Index, out Source, out
58             ↪ Target);
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         public Link(TLink index, TLink source, TLink target)
62         {
63             Index = index;
64             Source = source;
65             Target = target;
66         }
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         private static void SetValues(ref Link<TLink> other, out TLink index, out TLink source,
70             ↪ out TLink target)
71         {
72             index = other.Index;
73             source = other.Source;
74             target = other.Target;
75         }
76
77         [MethodImpl(MethodImplOptions.AggressiveInlining)]
78         private static void SetValues(IList<TLink> values, out TLink index, out TLink source,
79             ↪ out TLink target)
80         {
81             switch (values.Count)
82             {
83                 case 3:
84                     index = values[0];
85                     source = values[1];
86                     target = values[2];
87                     break;
88             }
89         }
90     }
91 }

```

```

82         case 2:
83             index = values[0];
84             source = values[1];
85             target = default;
86             break;
87         case 1:
88             index = values[0];
89             source = default;
90             target = default;
91             break;
92         default:
93             index = default;
94             source = default;
95             target = default;
96             break;
97     }
98 }
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 public override int GetHashCode() => (Index, Source, Target).GetHashCode();
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
105     && _equalityComparer.Equals(Source, _constants.Null)
106     && _equalityComparer.Equals(Target, _constants.Null);
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 public override bool Equals(object other) => other is Link<TLink> &&
110     => Equals((Link<TLink>)other);
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
114     && _equalityComparer.Equals(Source, other.Source)
115     && _equalityComparer.Equals(Target, other.Target);
116
117 [MethodImpl(MethodImplOptions.AggressiveInlining)]
118 public static string ToString(TLink index, TLink source, TLink target) => $"{index}:
119     {source}->{target}";
120
121 [MethodImpl(MethodImplOptions.AggressiveInlining)]
122 public static string ToString(TLink source, TLink target) => $"{source}->{target}";
123
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 public static implicit operator TLink[] (Link<TLink> link) => link.ToArray();
126
127 [MethodImpl(MethodImplOptions.AggressiveInlining)]
128 public static implicit operator Link<TLink>(TLink[] linkArray) => new
129     Link<TLink>(linkArray);
130
131 [MethodImpl(MethodImplOptions.AggressiveInlining)]
132 public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
133     ToString(Source, Target) : ToString(Index, Source, Target);
134
135 #region IList
136
137 public int Count
138 {
139     [MethodImpl(MethodImplOptions.AggressiveInlining)]
140     get => Length;
141 }
142
143 public bool IsReadOnly
144 {
145     [MethodImpl(MethodImplOptions.AggressiveInlining)]
146     get => true;
147 }
148
149 public TLink this[int index]
150 {
151     [MethodImpl(MethodImplOptions.AggressiveInlining)]
152     get
153     {
154         Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
155             => nameof(index));
156         if (index == _constants.IndexPart)
157         {
158             return Index;
159         }
160         if (index == _constants.SourcePart)
161         {

```

```

157         return Source;
158     }
159     if (index == _constants.TargetPart)
160     {
161         return Target;
162     }
163     throw new NotSupportedException(); // Impossible path due to
        ↪ Ensure.ArgumentInRange
164 }
165 [MethodImpl(MethodImplOptions.AggressiveInlining)]
166 set => throw new NotSupportedException();
167 }
168
169 [MethodImpl(MethodImplOptions.AggressiveInlining)]
170 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
171
172 [MethodImpl(MethodImplOptions.AggressiveInlining)]
173 public IEnumerator<TLink> GetEnumerator()
174 {
175     yield return Index;
176     yield return Source;
177     yield return Target;
178 }
179
180 [MethodImpl(MethodImplOptions.AggressiveInlining)]
181 public void Add(TLink item) => throw new NotSupportedException();
182
183 [MethodImpl(MethodImplOptions.AggressiveInlining)]
184 public void Clear() => throw new NotSupportedException();
185
186 [MethodImpl(MethodImplOptions.AggressiveInlining)]
187 public bool Contains(TLink item) => IndexOf(item) >= 0;
188
189 [MethodImpl(MethodImplOptions.AggressiveInlining)]
190 public void CopyTo(TLink[] array, int arrayIndex)
191 {
192     Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
193     Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
        ↪ nameof(arrayIndex));
194     if (arrayIndex + Length > array.Length)
195     {
196         throw new InvalidOperationException();
197     }
198     array[arrayIndex++] = Index;
199     array[arrayIndex++] = Source;
200     array[arrayIndex] = Target;
201 }
202
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 }

```

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

```

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

```

1.25 ./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 }

```

1.26 ./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 }

```

1.27 ./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 }

```

```

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 }

```

1.28 ./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 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Unsafe;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Memory
9  {
10     public struct LinksHeader<TLink> : IEquatable<LinksHeader<TLink>>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ⇨ EqualityComparer<TLink>.Default;
14
15         public static readonly long SizeInBytes = Structure<LinksHeader<TLink>>.Size;
16
17         public TLink AllocatedLinks;
18         public TLink ReservedLinks;
19         public TLink FreeLinks;
20         public TLink FirstFreeLink;
21         public TLink RootAsSource;
22         public TLink RootAsTarget;
23         public TLink LastFreeLink;
24         public TLink Reserved8;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public override bool Equals(object obj) => obj is LinksHeader<TLink> linksHeader ?
28             ⇨ Equals(linksHeader) : false;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public bool Equals(LinksHeader<TLink> other)
32             => _equalityComparer.Equals(AllocatedLinks, other.AllocatedLinks)
33             && _equalityComparer.Equals(ReservedLinks, other.ReservedLinks)
34             && _equalityComparer.Equals(FreeLinks, other.FreeLinks)
35             && _equalityComparer.Equals(FirstFreeLink, other.FirstFreeLink)
36             && _equalityComparer.Equals(RootAsSource, other.RootAsSource)
37             && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
38             && _equalityComparer.Equals(LastFreeLink, other.LastFreeLink)
39             && _equalityComparer.Equals(Reserved8, other.Reserved8);
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public override int GetHashCode() => (AllocatedLinks, ReservedLinks, FreeLinks,
43             ⇨ FirstFreeLink, RootAsSource, RootAsTarget, LastFreeLink, Reserved8).GetHashCode();
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public static bool operator ==(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
47             ⇨ left.Equals(right);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public static bool operator !=(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
51             ⇨ !(left == right);
52     }
53 }

```

```

47     }
48 }

```

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

```

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

```



```

66         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
67         ↪ secondLink.Source, secondLink.Target);
68     }
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
71     {
72         ref var firstLink = ref GetLinkDataPartReference(first);
73         ref var secondLink = ref GetLinkDataPartReference(second);
74         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
75         ↪ secondLink.Source, secondLink.Target);
76     }
77     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     }

```

```

138 // TODO: Return indices range instead of references count
139 [MethodImpl(MethodImplOptions.AggressiveInlining)]
140 public TLink CountUsages(TLink link)
141 {
142     var root = GetTreeRoot();
143     var total = GetSize(root);
144     var totalRightIgnore = Zero;
145     while (!EqualToZero(root))
146     {
147         var @base = GetBasePartValue(root);
148         if (LessOrEqualThan(@base, link))
149         {
150             root = GetRightOrDefault(root);
151         }
152         else
153         {
154             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
155             root = GetLeftOrDefault(root);
156         }
157     }
158     root = GetTreeRoot();
159     var totalLeftIgnore = Zero;
160     while (!EqualToZero(root))
161     {
162         var @base = GetBasePartValue(root);
163         if (GreaterOrEqualThan(@base, link))
164         {
165             root = GetLeftOrDefault(root);
166         }
167         else
168         {
169             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
170             root = GetRightOrDefault(root);
171         }
172     }
173     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
174 }
175
176 [MethodImpl(MethodImplOptions.AggressiveInlining)]
177 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
178     ↳ EachUsageCore(@base, GetTreeRoot(), handler);
179
180 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
181 ↳ low-level MSIL stack.
182 [MethodImpl(MethodImplOptions.AggressiveInlining)]
183 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
184 {
185     var @continue = Continue;
186     if (EqualToZero(link))
187     {
188         return @continue;
189     }
190     var linkBasePart = GetBasePartValue(link);
191     var @break = Break;
192     if (GreaterThan(linkBasePart, @base))
193     {
194         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
195         {
196             return @break;
197         }
198     }
199     else if (LessThan(linkBasePart, @base))
200     {
201         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
202         {
203             return @break;
204         }
205     }
206     else //if (linkBasePart == @base)
207     {
208         if (AreEqual(handler(GetLinkValues(link)), @break))
209         {
210             return @break;
211         }
212         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
213         {
214             return @break;
215         }
216         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))

```

```

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 }

```

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

```

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

```

```

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

```

1.32 ./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> :
    ↪ ExternalLinksSizeBalancedTreeMethodsBase<TLink>
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public ExternalLinksTargetsSizeBalancedTreeMethods(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() => 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 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,
47         ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
48         ↪ (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override void ClearNode(TLink node)
52     {
53         ref var link = ref GetLinkIndexPartReference(node);
54         link.LeftAsTarget = Zero;
55         link.RightAsTarget = Zero;
56         link.SizeAsTarget = Zero;
57     }
58 }

```

1.33 ./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)

```

```

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
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.

```

```

130 [MethodImpl(MethodImplOptions.AggressiveInlining)]
131 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
132 {
133     var @continue = Continue;
134     if (EqualToZero(link))
135     {
136         return @continue;
137     }
138     var @break = Break;
139     if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
140     {
141         return @break;
142     }
143     if (AreEqual(handler(GetLinkValues(link)), @break))
144     {
145         return @break;
146     }
147     if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
148     {
149         return @break;
150     }
151     return @continue;
152 }
153
154 [MethodImpl(MethodImplOptions.AggressiveInlining)]
155 protected override void PrintNodeValue(TLink node, StringBuilder sb)
156 {
157     ref var link = ref GetLinkDataPartReference(node);
158     sb.Append(' ');
159     sb.Append(link.Source);
160     sb.Append('-');
161     sb.Append('>');
162     sb.Append(link.Target);
163 }
164 }
165 }

```

1.34 ./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 }

```

```

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

```

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

```

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

```



```

36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected override TLink GetTreeRoot(TLink link) =>
38         ↪ GetLinkIndexPartReference(link).RootAsTarget;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override TLink GetBasePartValue(TLink link) =>
42         ↪ GetLinkDataPartReference(link).Target;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override TLink GetKeyPartValue(TLink link) =>
46         ↪ GetLinkDataPartReference(link).Source;
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     public override TLink Search(TLink source, TLink target) =>
58         ↪ SearchCore(GetTreeRoot(target), source);
59 }

```

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

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using static System.Runtime.CompilerServices.Unsafe;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Memory.Split.Generic
10 {
11     public unsafe class SplitMemoryLinks<TLink> : SplitMemoryLinksBase<TLink>
12     {
13         private readonly Func<ILinksTreeMethods<TLink>> _createInternalSourceTreeMethods;
14         private readonly Func<ILinksTreeMethods<TLink>> _createExternalSourceTreeMethods;
15         private readonly Func<ILinksTreeMethods<TLink>> _createInternalTargetTreeMethods;
16         private readonly Func<ILinksTreeMethods<TLink>> _createExternalTargetTreeMethods;
17         private byte* _header;
18         private byte* _linksDataParts;
19         private byte* _linksIndexParts;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
23             ↪ indexMemory) : this(dataMemory, indexMemory, DefaultLinksSizeStep) { }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
27             ↪ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
28             ↪ memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
32             ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
33             ↪ base(dataMemory, indexMemory, memoryReservationStep, constants)
34         {
35             _createInternalSourceTreeMethods = () => new
36                 ↪ InternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
37                 ↪ _linksIndexParts, _header);
38             _createExternalSourceTreeMethods = () => new
39                 ↪ ExternalLinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
40                 ↪ _linksIndexParts, _header);
41             _createInternalTargetTreeMethods = () => new
42                 ↪ InternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
43                 ↪ _linksIndexParts, _header);
44             _createExternalTargetTreeMethods = () => new
45                 ↪ ExternalLinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _linksDataParts,
46                 ↪ _linksIndexParts, _header);
47             Init(dataMemory, indexMemory);
48         }
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override void SetPointers(IResizableDirectMemory dataMemory,
52             ↪ IResizableDirectMemory indexMemory)

```

```

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

```

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

```

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

```

```

41     protected readonly long _indexMemoryReservationStepInBytes;
42
43     protected ILinksTreeMethods<TLink> InternalSourcesTreeMethods;
44     protected ILinksTreeMethods<TLink> ExternalSourcesTreeMethods;
45     protected ILinksTreeMethods<TLink> InternalTargetsTreeMethods;
46     protected ILinksTreeMethods<TLink> ExternalTargetsTreeMethods;
47     // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
48     ↪ нужно использовать не список а дерево, так как так можно быстрее проверить на
49     ↪ наличие связи внутри
50     protected ILinksListMethods<TLink> UnusedLinksListMethods;
51
52     /// <summary>
53     /// Возвращает общее число связей находящихся в хранилище.
54     /// </summary>
55     protected virtual TLink Total
56     {
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         get
59         {
60             ref var header = ref GetHeaderReference();
61             return Subtract(header.AllocatedLinks, header.FreeLinks);
62         }
63     }
64
65     public virtual LinksConstants<TLink> Constants
66     {
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         get;
69     }
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
73     ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants)
74     {
75         _dataMemory = dataMemory;
76         _indexMemory = indexMemory;
77         _dataMemoryReservationStepInBytes = memoryReservationStep * LinkDataPartSizeInBytes;
78         _indexMemoryReservationStepInBytes = memoryReservationStep *
79         ↪ LinkIndexPartSizeInBytes;
80         Constants = constants;
81     }
82
83     [MethodImpl(MethodImplOptions.AggressiveInlining)]
84     protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
85     ↪ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
86     ↪ memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }
87
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     protected virtual void Init(IResizableDirectMemory dataMemory, IResizableDirectMemory
90     ↪ indexMemory)
91     {
92         if (dataMemory.ReservedCapacity < _dataMemoryReservationStepInBytes)
93         {
94             dataMemory.ReservedCapacity = _dataMemoryReservationStepInBytes;
95         }
96         if (indexMemory.ReservedCapacity < _indexMemoryReservationStepInBytes)
97         {
98             indexMemory.ReservedCapacity = _indexMemoryReservationStepInBytes;
99         }
100         SetPointers(dataMemory, indexMemory);
101         ref var header = ref GetHeaderReference();
102         // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
103         // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
104         dataMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
105         ↪ LinkDataPartSizeInBytes) + LinkDataPartSizeInBytes; // First link is read only
106         ↪ zero link.
107         indexMemory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) *
108         ↪ LinkIndexPartSizeInBytes) + LinkHeaderSizeInBytes;
109         // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
110         // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
111         header.ReservedLinks = ConvertToAddress((dataMemory.ReservedCapacity -
112         ↪ LinkDataPartSizeInBytes) / LinkDataPartSizeInBytes);
113     }
114
115     [MethodImpl(MethodImplOptions.AggressiveInlining)]
116     public virtual TLink Count(IList<TLink> restrictions)
117     {
118         // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
119         if (restrictions.Count == 0)

```

```

109 {
110     return Total;
111 }
112 var constants = Constants;
113 var any = constants.Any;
114 var index = restrictions[constants.IndexPart];
115 if (restrictions.Count == 1)
116 {
117     if (AreEqual(index, any))
118     {
119         return Total;
120     }
121     return Exists(index) ? GetOne() : GetZero();
122 }
123 if (restrictions.Count == 2)
124 {
125     var value = restrictions[1];
126     if (AreEqual(index, any))
127     {
128         if (AreEqual(value, any))
129         {
130             return Total; // Any - как отсутствие ограничения
131         }
132         var externalReferencesRange = constants.ExternalReferencesRange;
133         if (externalReferencesRange.HasValue &&
134             ⇨ externalReferencesRange.Value.Contains(value))
135         {
136             return Add(ExternalSourcesTreeMethods.CountUsages(value),
137                 ⇨ ExternalTargetsTreeMethods.CountUsages(value));
138         }
139         else
140         {
141             return Add(InternalSourcesTreeMethods.CountUsages(value),
142                 ⇨ InternalTargetsTreeMethods.CountUsages(value));
143         }
144     }
145     else
146     {
147         if (!Exists(index))
148         {
149             return GetZero();
150         }
151         if (AreEqual(value, any))
152         {
153             return GetOne();
154         }
155         ref var storedLinkValue = ref GetLinkDataPartReference(index);
156         if (AreEqual(storedLinkValue.Source, value) ||
157             ⇨ AreEqual(storedLinkValue.Target, value))
158         {
159             return GetOne();
160         }
161         return GetZero();
162     }
163 }
164 if (restrictions.Count == 3)
165 {
166     var externalReferencesRange = constants.ExternalReferencesRange;
167     var source = restrictions[constants.SourcePart];
168     var target = restrictions[constants.TargetPart];
169     if (AreEqual(index, any))
170     {
171         if (AreEqual(source, any) && AreEqual(target, any))
172         {
173             return Total;
174         }
175         else if (AreEqual(source, any))
176         {
177             if (externalReferencesRange.HasValue &&
178                 ⇨ externalReferencesRange.Value.Contains(target))
179             {
180                 return ExternalTargetsTreeMethods.CountUsages(target);
181             }
182             else
183             {
184                 return InternalTargetsTreeMethods.CountUsages(target);
185             }
186         }
187     }
188 }

```

```

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

```

```

255     }
256     var value = default(TLink);
257     if (AreEqual(source, any))
258     {
259         value = target;
260     }
261     if (AreEqual(target, any))
262     {
263         value = source;
264     }
265     if (AreEqual(storedLinkValue.Source, value) ||
        ↪ AreEqual(storedLinkValue.Target, value))
266     {
267         return GetOne();
268     }
269     return GetZero();
270 }
271 }
272 throw new NotSupportedException("Другие размеры и способы ограничений не
        ↪ поддерживаются.");
273 }
274
275 [MethodImpl(MethodImplOptions.AggressiveInlining)]
276 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
277 {
278     var constants = Constants;
279     var @break = constants.Break;
280     if (restrictions.Count == 0)
281     {
282         for (var link = GetOne(); LessOrEqualThan(link,
            ↪ GetHeaderReference().AllocatedLinks); link = Increment(link))
283         {
284             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
285             {
286                 return @break;
287             }
288         }
289         return @break;
290     }
291     var @continue = constants.Continue;
292     var any = constants.Any;
293     var index = restrictions[constants.IndexPart];
294     if (restrictions.Count == 1)
295     {
296         if (AreEqual(index, any))
297         {
298             return Each(handler, Array.Empty<TLink>());
299         }
300         if (!Exists(index))
301         {
302             return @continue;
303         }
304         return handler(GetLinkStruct(index));
305     }
306     if (restrictions.Count == 2)
307     {
308         var value = restrictions[1];
309         if (AreEqual(index, any))
310         {
311             if (AreEqual(value, any))
312             {
313                 return Each(handler, Array.Empty<TLink>());
314             }
315             if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
316             {
317                 return @break;
318             }
319             return Each(handler, new Link<TLink>(index, any, value));
320         }
321         else
322         {
323             if (!Exists(index))
324             {
325                 return @continue;
326             }
327             if (AreEqual(value, any))
328             {
329                 return handler(GetLinkStruct(index));

```

```

330     }
331     ref var storedLinkValue = ref GetLinkDataPartReference(index);
332     if (AreEqual(storedLinkValue.Source, value) ||
333         AreEqual(storedLinkValue.Target, value))
334     {
335         return handler(GetLinkStruct(index));
336     }
337     return @continue;
338 }
339 }
340 if (restrictions.Count == 3)
341 {
342     var externalReferencesRange = constants.ExternalReferencesRange;
343     var source = restrictions[constants.SourcePart];
344     var target = restrictions[constants.TargetPart];
345     if (AreEqual(index, any))
346     {
347         if (AreEqual(source, any) && AreEqual(target, any))
348         {
349             return Each(handler, Array.Empty<TLink>());
350         }
351         else if (AreEqual(source, any))
352         {
353             if (externalReferencesRange.HasValue &&
354                 ⇨ externalReferencesRange.Value.Contains(target))
355             {
356                 return ExternalTargetsTreeMethods.EachUsage(target, handler);
357             }
358             else
359             {
360                 return InternalTargetsTreeMethods.EachUsage(target, handler);
361             }
362         }
363         else if (AreEqual(target, any))
364         {
365             if (externalReferencesRange.HasValue &&
366                 ⇨ externalReferencesRange.Value.Contains(source))
367             {
368                 return ExternalSourcesTreeMethods.EachUsage(source, handler);
369             }
370             else
371             {
372                 return InternalSourcesTreeMethods.EachUsage(source, handler);
373             }
374         }
375         else //if(source != Any && target != Any)
376         {
377             TLink link;
378             if (externalReferencesRange.HasValue)
379             {
380                 if (externalReferencesRange.Value.Contains(source) &&
381                     ⇨ externalReferencesRange.Value.Contains(target))
382                 {
383                     link = ExternalSourcesTreeMethods.Search(source, target);
384                 }
385                 else if (externalReferencesRange.Value.Contains(source))
386                 {
387                     link = InternalTargetsTreeMethods.Search(source, target);
388                 }
389                 else if (externalReferencesRange.Value.Contains(target))
390                 {
391                     link = InternalSourcesTreeMethods.Search(source, target);
392                 }
393                 else
394                 {
395                     if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
396                         ⇨ InternalTargetsTreeMethods.CountUsages(target)))
397                     {
398                         link = InternalTargetsTreeMethods.Search(source, target);
399                     }
400                     else
401                     {
402                         link = InternalSourcesTreeMethods.Search(source, target);
403                     }
404                 }
405             }
406             else
407             {
408                 link = InternalSourcesTreeMethods.Search(source, target);
409             }
410         }
411     }
412 }

```

```

404         if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
405             ↪ InternalTargetsTreeMethods.CountUsages(target)))
406         {
407             link = InternalTargetsTreeMethods.Search(source, target);
408         }
409         else
410         {
411             link = InternalSourcesTreeMethods.Search(source, target);
412         }
413         return AreEqual(link, constants.Null) ? @continue :
414             ↪ handler(GetLinkStruct(link));
415     }
416     else
417     {
418         if (!Exists(index))
419         {
420             return @continue;
421         }
422         if (AreEqual(source, any) && AreEqual(target, any))
423         {
424             return handler(GetLinkStruct(index));
425         }
426         ref var storedLinkValue = ref GetLinkDataPartReference(index);
427         if (!AreEqual(source, any) && !AreEqual(target, any))
428         {
429             if (AreEqual(storedLinkValue.Source, source) &&
430                 AreEqual(storedLinkValue.Target, target))
431             {
432                 return handler(GetLinkStruct(index));
433             }
434             return @continue;
435         }
436         var value = default(TLink);
437         if (AreEqual(source, any))
438         {
439             value = target;
440         }
441         if (AreEqual(target, any))
442         {
443             value = source;
444         }
445         if (AreEqual(storedLinkValue.Source, value) ||
446             AreEqual(storedLinkValue.Target, value))
447         {
448             return handler(GetLinkStruct(index));
449         }
450         return @continue;
451     }
452 }
453 throw new NotSupportedException("Другие размеры и способы ограничений не
454     ↪ поддерживаются.");
455 }
456 /// <remarks>
457 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
458     ↪ в другом месте (но не в менеджере памяти, а в логике Links)
459 /// </remarks>
460 [MethodImpl(MethodImplOptions.AggressiveInlining)]
461 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
462 {
463     var constants = Constants;
464     var @null = constants.Null;
465     var externalReferencesRange = constants.ExternalReferencesRange;
466     var linkIndex = restrictions[constants.IndexPart];
467     ref var link = ref GetLinkDataPartReference(linkIndex);
468     var source = link.Source;
469     var target = link.Target;
470     ref var header = ref GetHeaderReference();
471     ref var rootAsSource = ref header.RootAsSource;
472     ref var rootAsTarget = ref header.RootAsTarget;
473     // Будет корректно работать только в том случае, если пространство выделенной связи
474     ↪ предварительно заполнено нулями
475     if (!AreEqual(source, @null))
476     {
477         if (externalReferencesRange.HasValue &&
478             ↪ externalReferencesRange.Value.Contains(source))

```



```

476         {
477             ExternalSourcesTreeMethods.Detach(ref rootAsSource, linkIndex);
478         }
479         else
480         {
481             InternalSourcesTreeMethods.Detach(ref
482                 ↪ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
483         }
484     if (!AreEqual(target, @null))
485     {
486         if (externalReferencesRange.HasValue &&
487             ↪ externalReferencesRange.Value.Contains(target))
488         {
489             ExternalTargetsTreeMethods.Detach(ref rootAsTarget, linkIndex);
490         }
491         else
492         {
493             InternalTargetsTreeMethods.Detach(ref
494                 ↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
495         }
496     }
497     source = link.Source = substitution[constants.SourcePart];
498     target = link.Target = substitution[constants.TargetPart];
499     if (!AreEqual(source, @null))
500     {
501         if (externalReferencesRange.HasValue &&
502             ↪ externalReferencesRange.Value.Contains(source))
503         {
504             ExternalSourcesTreeMethods.Attach(ref rootAsSource, linkIndex);
505         }
506         else
507         {
508             InternalSourcesTreeMethods.Attach(ref
509                 ↪ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
510         }
511     }
512     if (!AreEqual(target, @null))
513     {
514         if (externalReferencesRange.HasValue &&
515             ↪ externalReferencesRange.Value.Contains(target))
516         {
517             ExternalTargetsTreeMethods.Attach(ref rootAsTarget, linkIndex);
518         }
519         else
520         {
521             InternalTargetsTreeMethods.Attach(ref
522                 ↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
523         }
524     }
525     return linkIndex;
526 }
527
528 /// <remarks>
529 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
530 ↪ пространство
531 /// </remarks>
532 [MethodImpl(MethodImplOptions.AggressiveInlining)]
533 public virtual TLink Create(IList<TLink> restrictions)
534 {
535     ref var header = ref GetHeaderReference();
536     var freeLink = header.FirstFreeLink;
537     if (!AreEqual(freeLink, Constants.Null))
538     {
539         UnusedLinksListMethods.Detach(freeLink);
540     }
541     else
542     {
543         var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
544         if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
545         {
546             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
547         }
548         if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
549         {
550             _dataMemory.ReservedCapacity += _dataMemory.ReservationStepInBytes;
551             _indexMemory.ReservedCapacity += _indexMemory.ReservationStepInBytes;
552             SetPointers(_dataMemory, _indexMemory);
553         }
554     }
555 }

```

```

546         header = ref GetHeaderReference();
547         header.ReservedLinks = ConvertToAddress(_dataMemory.ReservedCapacity /
        ↪ LinkDataPartSizeInBytes);
548     }
549     header.AllocatedLinks = Increment(header.AllocatedLinks);
550     _dataMemory.UsedCapacity += LinkDataPartSizeInBytes;
551     _indexMemory.UsedCapacity += LinkIndexPartSizeInBytes;
552     freeLink = header.AllocatedLinks;
553 }
554 return freeLink;
555 }
556
557 [MethodImpl(MethodImplOptions.AggressiveInlining)]
558 public virtual void Delete(ICollection<TLink> restrictions)
559 {
560     ref var header = ref GetHeaderReference();
561     var link = restrictions[Constants.IndexPart];
562     if (LessThan(link, header.AllocatedLinks))
563     {
564         UnusedLinksListMethods.AttachAsFirst(link);
565     }
566     else if (AreEqual(link, header.AllocatedLinks))
567     {
568         header.AllocatedLinks = Decrement(header.AllocatedLinks);
569         _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
570         _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
571         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
572         ↪ пока не дойдём до первой существующей связи
573         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
574         while (GreaterThan(header.AllocatedLinks, GetZero()) &&
575             ↪ IsUnusedLink(header.AllocatedLinks))
576         {
577             UnusedLinksListMethods.Detach(header.AllocatedLinks);
578             header.AllocatedLinks = Decrement(header.AllocatedLinks);
579             _dataMemory.UsedCapacity -= LinkDataPartSizeInBytes;
580             _indexMemory.UsedCapacity -= LinkIndexPartSizeInBytes;
581         }
582     }
583 }
584
585 [MethodImpl(MethodImplOptions.AggressiveInlining)]
586 public IList<TLink> GetLinkStruct(TLink linkIndex)
587 {
588     ref var link = ref GetLinkDataPartReference(linkIndex);
589     return new Link<TLink>(linkIndex, link.Source, link.Target);
590 }
591
592 /// <remarks>
593 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
594 ↪ адрес реально поменялся
595 ///
596 /// Указатель this.links может быть в том же месте,
597 /// так как 0-я связь не используется и имеет такой же размер как Header,
598 /// поэтому header размещается в том же месте, что и 0-я связь
599 /// </remarks>
600 [MethodImpl(MethodImplOptions.AggressiveInlining)]
601 protected abstract void SetPointers(IResizableDirectMemory dataMemory,
602     ↪ IResizableDirectMemory indexMemory);
603
604 [MethodImpl(MethodImplOptions.AggressiveInlining)]
605 protected virtual void ResetPointers()
606 {
607     InternalSourcesTreeMethods = null;
608     ExternalSourcesTreeMethods = null;
609     InternalTargetsTreeMethods = null;
610     ExternalTargetsTreeMethods = null;
611     UnusedLinksListMethods = null;
612 }
613
614 [MethodImpl(MethodImplOptions.AggressiveInlining)]
615 protected abstract ref LinksHeader<TLink> GetHeaderReference();
616
617 [MethodImpl(MethodImplOptions.AggressiveInlining)]
618 protected abstract ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex);
619
620 [MethodImpl(MethodImplOptions.AggressiveInlining)]
621 protected abstract ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
622     ↪ linkIndex);

```

```

619 [MethodImpl(MethodImplOptions.AggressiveInlining)]
620 protected virtual bool Exists(TLink link)
621     => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
622     && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
623     && !IsUnusedLink(link);
624
625 [MethodImpl(MethodImplOptions.AggressiveInlining)]
626 protected virtual bool IsUnusedLink(TLink linkIndex)
627 {
628     if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
629         ↪ is not needed
630     {
631         // TODO: Reduce access to memory in different location (should be enough to use
632         ↪ just linkIndexPart)
633         ref var linkDataPart = ref GetLinkDataPartReference(linkIndex);
634         ref var linkIndexPart = ref GetLinkIndexPartReference(linkIndex);
635         return AreEqual(linkIndexPart.SizeAsSource, default) &&
636             ↪ !AreEqual(linkDataPart.Source, default);
637     }
638     else
639     {
640         return true;
641     }
642 }
643
644 [MethodImpl(MethodImplOptions.AggressiveInlining)]
645 protected virtual TLink GetOne() => _one;
646
647 [MethodImpl(MethodImplOptions.AggressiveInlining)]
648 protected virtual TLink GetZero() => default;
649
650 [MethodImpl(MethodImplOptions.AggressiveInlining)]
651 protected virtual bool AreEqual(TLink first, TLink second) =>
652     ↪ _equalityComparer.Equals(first, second);
653
654 [MethodImpl(MethodImplOptions.AggressiveInlining)]
655 protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
656     ↪ second) < 0;
657
658 [MethodImpl(MethodImplOptions.AggressiveInlining)]
659 protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
660     ↪ _comparer.Compare(first, second) <= 0;
661
662 [MethodImpl(MethodImplOptions.AggressiveInlining)]
663 protected virtual bool GreaterThan(TLink first, TLink second) =>
664     ↪ _comparer.Compare(first, second) > 0;
665
666 [MethodImpl(MethodImplOptions.AggressiveInlining)]
667 protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
668     ↪ _comparer.Compare(first, second) >= 0;
669
670 [MethodImpl(MethodImplOptions.AggressiveInlining)]
671 protected virtual long ConvertToInt64(TLink value) =>
672     ↪ _addressToInt64Converter.Convert(value);
673
674 [MethodImpl(MethodImplOptions.AggressiveInlining)]
675 protected virtual TLink ConvertToAddress(long value) =>
676     ↪ _int64ToAddressConverter.Convert(value);
677
678 [MethodImpl(MethodImplOptions.AggressiveInlining)]
679 protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
680     ↪ second);
681
682 [MethodImpl(MethodImplOptions.AggressiveInlining)]
683 protected virtual TLink Subtract(TLink first, TLink second) =>
684     ↪ Arithmetic<TLink>.Subtract(first, second);
685
686 [MethodImpl(MethodImplOptions.AggressiveInlining)]
687 protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
688
689 [MethodImpl(MethodImplOptions.AggressiveInlining)]
690 protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
691
692 #region Disposable
693
694 protected override bool AllowMultipleDisposeCalls
695 {
696     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

685         get => true;
686     }
687
688     [MethodImpl(MethodImplOptions.AggressiveInlining)]
689     protected override void Dispose(bool manual, bool wasDisposed)
690     {
691         if (!wasDisposed)
692         {
693             ResetPointers();
694             _dataMemory.DisposeIfPossible();
695             _indexMemory.DisposeIfPossible();
696         }
697     }
698
699     #endregion
700 }
701 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Methods.Lists;
3  using Platform.Converters;
4  using static System.Runtime.CompilerServices.Unsafe;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Memory.Split.Generic
9  {
10     public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
11         ↳ ILinksListMethods<TLink>
12     {
13         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
14             ↳ UncheckedConverter<TLink, long>.Default;
15
16         private readonly byte* _links;
17         private readonly byte* _header;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnusedLinksListMethods(byte* links, byte* header)
21         {
22             _links = links;
23             _header = header;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
28             ↳ AsRef<LinksHeader<TLink>>(_header);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected virtual ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) => ref
32             ↳ AsRef<RawLinkDataPart<TLink>>(_links + (RawLinkDataPart<TLink>.SizeInBytes *
33             ↳ _addressToInt64Converter.Convert(link)));
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected override TLink GetPrevious(TLink element) =>
43             ↳ GetLinkDataPartReference(element).Source;
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override TLink GetNext(TLink element) =>
47             ↳ GetLinkDataPartReference(element).Target;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override TLink GetSize() => GetHeaderReference().FreeLinks;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
54             ↳ element;
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
58             ↳ element;
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

52     protected override void SetPrevious(TLink element, TLink previous) =>
53         ↪ GetLinkDataPartReference(element).Source = previous;
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override void SetNext(TLink element, TLink next) =>
57         ↪ GetLinkDataPartReference(element).Target = next;
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
61 }
62 }

```

1.39 ./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 }

```

1.40 ./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 }

```

```

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

```

1.41 ./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 }

```

1.42 ./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 :
    ↪ UInt32ExternalLinksSizeBalancedTreeMethodsBase
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public UInt32ExternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
    ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
    ↪ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
    ↪ linksIndexParts, header) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected override ref TLink GetLeftReference(TLink node) => ref
    ↪ LinksIndexParts[node].LeftAsSource;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

1.43 ./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)]

```



```

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

```

1.44 ./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)]

```

```

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
44     ↪ always true for ulong
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override bool LessOrEqualThanZero(TLink value) => value == 0UL; // value is
48     ↪ always >= 0 for ulong
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override bool LessThanZero(TLink value) => false; // value < 0 is always false
55     ↪ for ulong
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected override bool LessThan(TLink first, TLink second) => first < second;
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected override TLink Increment(TLink value) => ++value;
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override TLink Decrement(TLink value) => --value;
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override TLink Add(TLink first, TLink second) => first + second;
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override TLink Subtract(TLink first, TLink second) => first - second;
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink link) =>
74     ↪ ref LinksDataParts[link];
75
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink link) =>
78     ↪ ref LinksIndexParts[link];
79
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     protected override bool FirstIsToLeftOfSecond(TLink first, TLink second) =>
82     ↪ GetKeyPartValue(first) < GetKeyPartValue(second);
83
84     [MethodImpl(MethodImplOptions.AggressiveInlining)]
85     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second) =>
86     ↪ GetKeyPartValue(first) > GetKeyPartValue(second);
87
88 }
89
90 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt32;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      public unsafe class UInt32InternalLinksSourcesSizeBalancedTreeMethods :
9      ↪ UInt32InternalLinksSizeBalancedTreeMethodsBase
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public UInt32InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
13         ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
14         ↪ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
15         ↪ linksIndexParts, header) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

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

```

1  using System.Runtime.CompilerServices;
2  using TLink = System.UInt32;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Memory.Split.Specific
7  {
8      public unsafe class UInt32InternalLinksTargetsSizeBalancedTreeMethods :
9          ↳ UInt32InternalLinksSizeBalancedTreeMethodsBase
10      {
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          public UInt32InternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink>
13              ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
14              ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
15              ↳ linksIndexParts, header) { }
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override ref TLink GetLeftReference(TLink node) => ref
19              ↳ LinksIndexParts[node].LeftAsTarget;
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          protected override ref TLink GetRightReference(TLink node) => ref
23              ↳ LinksIndexParts[node].RightAsTarget;
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsTarget;
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsTarget;
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          protected override void SetLeft(TLink node, TLink left) =>
33              ↳ LinksIndexParts[node].LeftAsTarget = left;
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected override void SetRight(TLink node, TLink right) =>
37              ↳ LinksIndexParts[node].RightAsTarget = right;
38
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
41
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          protected override void SetSize(TLink node, TLink size) =>
44              ↳ LinksIndexParts[node].SizeAsTarget = size;
45
46          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47          protected override TLink GetTreeRoot(TLink node) => LinksIndexParts[node].RootAsTarget;
48
49          [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.LeftAsTarget = Zero;
60              link.RightAsTarget = Zero;
61              link.SizeAsTarget = Zero;
62          }
63
64          public override TLink Search(TLink source, TLink target) =>
65              ↳ SearchCore(GetTreeRoot(source), target);
66      }
67 }

```

```

22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsTarget;
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     protected override void SetLeft(TLink node, TLink left) =>
27         ↳ LinksIndexParts[node].LeftAsTarget = left;
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     protected override void SetRight(TLink node, TLink right) =>
31         ↳ LinksIndexParts[node].RightAsTarget = right;
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsTarget;
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected override void SetSize(TLink node, TLink size) =>
38         ↳ LinksIndexParts[node].SizeAsTarget = size;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected override TLink GetTreeRoot(TLink node) => LinksIndexParts[node].RootAsTarget;
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Target;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Source;
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override void ClearNode(TLink node)
51     {
52         ref var link = ref LinksIndexParts[node];
53         link.LeftAsTarget = Zero;
54         link.RightAsTarget = Zero;
55         link.SizeAsTarget = Zero;
56     }
57
58     public override TLink Search(TLink source, TLink target) =>
59         ↳ SearchCore(GetTreeRoot(target), source);
60 }

```

1.47 ./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
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public UInt32SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
33             ↳ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
34             ↳ base(dataMemory, indexMemory, memoryReservationStep, constants)
35         {
36             _createInternalSourceTreeMethods = () => new
37                 ↳ UInt32InternalLinksSourcesSizeBalancedTreeMethods(Constants, _linksDataParts,
38                 ↳ _linksIndexParts, _header);
39         }
40     }
41 }

```

```

32     _createExternalSourceTreeMethods = () => new
33     ↪ UInt32ExternalLinksSourcesSizeBalancedTreeMethods(Constants, _linksDataParts,
34     ↪ _linksIndexParts, _header);
35     _createInternalTargetTreeMethods = () => new
36     ↪ UInt32InternalLinksTargetsSizeBalancedTreeMethods(Constants, _linksDataParts,
37     ↪ _linksIndexParts, _header);
38     _createExternalTargetTreeMethods = () => new
39     ↪ UInt32ExternalLinksTargetsSizeBalancedTreeMethods(Constants, _linksDataParts,
40     ↪ _linksIndexParts, _header);
41     Init(dataMemory, indexMemory);
42 }
43
44 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 protected override void SetPointers(IResizableDirectMemory dataMemory,
46 ↪ IResizableDirectMemory indexMemory)
47 {
48     _linksDataParts = (RawLinkDataPart<TLink>*)dataMemory.Pointer;
49     _linksIndexParts = (RawLinkIndexPart<TLink>*)indexMemory.Pointer;
50     _header = (LinksHeader<TLink>*)indexMemory.Pointer;
51     InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
52     ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
53     InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
54     ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
55     UnusedLinksListMethods = new UInt32UnusedLinksListMethods(_linksDataParts, _header);
56 }
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 protected override void ResetPointers()
60 {
61     base.ResetPointers();
62     _linksDataParts = null;
63     _linksIndexParts = null;
64     _header = null;
65 }
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
72 ↪ => ref _linksDataParts[linkIndex];
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
76 ↪ linkIndex) => ref _linksIndexParts[linkIndex];
77
78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 protected override bool AreEqual(TLink first, TLink second) => first == second;
80
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 protected override bool LessThan(TLink first, TLink second) => first < second;
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 protected override bool LessOrEqualThan(TLink first, TLink second) => first <= second;
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 protected override bool GreaterThan(TLink first, TLink second) => first > second;
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 protected override bool GreaterOrEqualThan(TLink first, TLink second) => first >= second;
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 protected override TLink GetZero() => 0U;
95
96 [MethodImpl(MethodImplOptions.AggressiveInlining)]
97 protected override TLink GetOne() => 1U;
98
99 [MethodImpl(MethodImplOptions.AggressiveInlining)]
100 protected override long ConvertToInt64(TLink value) => value;
101
102 [MethodImpl(MethodImplOptions.AggressiveInlining)]
103 protected override TLink ConvertToAddress(long value) => (TLink)value;
104
105 [MethodImpl(MethodImplOptions.AggressiveInlining)]
106 protected override TLink Add(TLink first, TLink second) => first + second;
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 protected override TLink Subtract(TLink first, TLink second) => first - second;

```

```

102     [MethodImpl(MethodImplOptions.AggressiveInlining)]
103     protected override TLink Increment(TLink link) => ++link;
104
105     [MethodImpl(MethodImplOptions.AggressiveInlining)]
106     protected override TLink Decrement(TLink link) => --link;
107 }
108 }

```

1.48 ./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 }

```

1.49 ./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 }

```

```

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

```

1.50 ./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)]

```

```

11 public UInt64ExternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
    ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
    ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
    ↳ linksIndexParts, header) { }
12
13 [MethodImpl(MethodImplOptions.AggressiveInlining)]
14 protected override ref TLink GetLeftReference(TLink node) => ref
    ↳ LinksIndexParts[node].LeftAsSource;
15
16 [MethodImpl(MethodImplOptions.AggressiveInlining)]
17 protected override ref TLink GetRightReference(TLink node) => ref
    ↳ LinksIndexParts[node].RightAsSource;
18
19 [MethodImpl(MethodImplOptions.AggressiveInlining)]
20 protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
21
22 [MethodImpl(MethodImplOptions.AggressiveInlining)]
23 protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
24
25 [MethodImpl(MethodImplOptions.AggressiveInlining)]
26 protected override void SetLeft(TLink node, TLink left) =>
    ↳ LinksIndexParts[node].LeftAsSource = left;
27
28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 protected override void SetRight(TLink node, TLink right) =>
    ↳ LinksIndexParts[node].RightAsSource = right;
30
31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
33
34 [MethodImpl(MethodImplOptions.AggressiveInlining)]
35 protected override void SetSize(TLink node, TLink size) =>
    ↳ LinksIndexParts[node].SizeAsSource = size;
36
37 [MethodImpl(MethodImplOptions.AggressiveInlining)]
38 protected override TLink GetTreeRoot() => Header->RootAsSource;
39
40 [MethodImpl(MethodImplOptions.AggressiveInlining)]
41 protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
42
43 [MethodImpl(MethodImplOptions.AggressiveInlining)]
44 protected override bool FirstIsToLeftOfSecond(TLink firstSource, TLink firstTarget,
    ↳ TLink secondSource, TLink secondTarget)
45     => firstSource < secondSource || firstSource == secondSource && firstTarget <
    ↳ secondTarget;
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
    ↳ TLink secondSource, TLink secondTarget)
49     => firstSource > secondSource || firstSource == secondSource && firstTarget >
    ↳ secondTarget;
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override void ClearNode(TLink node)
53 {
54     ref var link = ref LinksIndexParts[node];
55     link.LeftAsSource = Zero;
56     link.RightAsSource = Zero;
57     link.SizeAsSource = Zero;
58 }
59 }
60 }

```

1.51 ./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>
            ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
            ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
            ↳ linksIndexParts, header) { }

```



```

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

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Data.Doublets.Memory.Split.Generic;
3 using TLink = System.UInt64;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Memory.Split.Specific
8 {
9     public unsafe abstract class UInt64InternalLinksSizeBalancedTreeMethodsBase :
10     ↪ InternalLinksSizeBalancedTreeMethodsBase<TLink>
11     {
12         protected new readonly RawLinkDataPart<TLink>* LinksDataParts;
13         protected new readonly RawLinkIndexPart<TLink>* LinksIndexParts;
14         protected new readonly LinksHeader<TLink>* Header;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected UInt64InternalLinksSizeBalancedTreeMethodsBase(LinksConstants<TLink>
18         ↪ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
19         ↪ linksIndexParts, LinksHeader<TLink>* header)

```

```

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

```

1.53 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesSizeBalancedTreeMethod

```

1 using System.Runtime.CompilerServices;
2 using TLink = System.UInt64;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Memory.Split.Specific

```

```

7 {
8     public unsafe class UInt64InternalLinksSourcesSizeBalancedTreeMethods :
9         ↳ UInt64InternalLinksSizeBalancedTreeMethodsBase
10    {
11        [MethodImpl(MethodImplOptions.AggressiveInlining)]
12        public UInt64InternalLinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink>
13            ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
14            ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
15            ↳ linksIndexParts, header) { }
16
17        [MethodImpl(MethodImplOptions.AggressiveInlining)]
18        protected override ref TLink GetLeftReference(TLink node) => ref
19            ↳ LinksIndexParts[node].LeftAsSource;
20
21        [MethodImpl(MethodImplOptions.AggressiveInlining)]
22        protected override ref TLink GetRightReference(TLink node) => ref
23            ↳ LinksIndexParts[node].RightAsSource;
24
25        [MethodImpl(MethodImplOptions.AggressiveInlining)]
26        protected override TLink GetLeft(TLink node) => LinksIndexParts[node].LeftAsSource;
27
28        [MethodImpl(MethodImplOptions.AggressiveInlining)]
29        protected override TLink GetRight(TLink node) => LinksIndexParts[node].RightAsSource;
30
31        [MethodImpl(MethodImplOptions.AggressiveInlining)]
32        protected override void SetLeft(TLink node, TLink left) =>
33            ↳ LinksIndexParts[node].LeftAsSource = left;
34
35        [MethodImpl(MethodImplOptions.AggressiveInlining)]
36        protected override void SetRight(TLink node, TLink right) =>
37            ↳ LinksIndexParts[node].RightAsSource = right;
38
39        [MethodImpl(MethodImplOptions.AggressiveInlining)]
40        protected override TLink GetSize(TLink node) => LinksIndexParts[node].SizeAsSource;
41
42        [MethodImpl(MethodImplOptions.AggressiveInlining)]
43        protected override void SetSize(TLink node, TLink size) =>
44            ↳ LinksIndexParts[node].SizeAsSource = size;
45
46        [MethodImpl(MethodImplOptions.AggressiveInlining)]
47        protected override TLink GetTreeRoot(TLink node) => LinksIndexParts[node].RootAsSource;
48
49        [MethodImpl(MethodImplOptions.AggressiveInlining)]
50        protected override TLink GetBasePartValue(TLink node) => LinksDataParts[node].Source;
51
52        [MethodImpl(MethodImplOptions.AggressiveInlining)]
53        protected override TLink GetKeyPartValue(TLink node) => LinksDataParts[node].Target;
54
55        [MethodImpl(MethodImplOptions.AggressiveInlining)]
56        protected override void ClearNode(TLink node)
57        {
58            ref var link = ref LinksIndexParts[node];
59            link.LeftAsSource = Zero;
60            link.RightAsSource = Zero;
61            link.SizeAsSource = Zero;
62        }
63
64        public override TLink Search(TLink source, TLink target) =>
65            ↳ SearchCore(GetTreeRoot(source), target);
66    }
67 }

```

1.54 ./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksTargetsSizeBalancedTreeMethods

```

1 using System.Runtime.CompilerServices;
2 using TLink = System.UInt64;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Memory.Split.Specific
7 {
8     public unsafe class UInt64InternalLinksTargetsSizeBalancedTreeMethods :
9         ↳ UInt64InternalLinksSizeBalancedTreeMethodsBase
10    {
11        [MethodImpl(MethodImplOptions.AggressiveInlining)]
12        public UInt64InternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink>
13            ↳ constants, RawLinkDataPart<TLink>* linksDataParts, RawLinkIndexPart<TLink>*
14            ↳ linksIndexParts, LinksHeader<TLink>* header) : base(constants, linksDataParts,
15            ↳ linksIndexParts, header) { }
16
17        [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

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

```

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

```

```

26 public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
    ↳ memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }
27
28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 public UInt64SplitMemoryLinks(IResizableDirectMemory dataMemory, IResizableDirectMemory
    ↳ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants) :
    ↳ base(dataMemory, indexMemory, memoryReservationStep, constants)
30 {
31     _createInternalSourceTreeMethods = () => new
        ↳ UInt64InternalLinksSourcesSizeBalancedTreeMethods(Constants, _linksDataParts,
        ↳ _linksIndexParts, _header);
32     _createExternalSourceTreeMethods = () => new
        ↳ UInt64ExternalLinksSourcesSizeBalancedTreeMethods(Constants, _linksDataParts,
        ↳ _linksIndexParts, _header);
33     _createInternalTargetTreeMethods = () => new
        ↳ UInt64InternalLinksTargetsSizeBalancedTreeMethods(Constants, _linksDataParts,
        ↳ _linksIndexParts, _header);
34     _createExternalTargetTreeMethods = () => new
        ↳ UInt64ExternalLinksTargetsSizeBalancedTreeMethods(Constants, _linksDataParts,
        ↳ _linksIndexParts, _header);
35     Init(dataMemory, indexMemory);
36 }
37
38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 protected override void SetPointers(IResizableDirectMemory dataMemory,
    ↳ IResizableDirectMemory indexMemory)
40 {
41     _linksDataParts = (RawLinkDataPart<TLink>*)dataMemory.Pointer;
42     _linksIndexParts = (RawLinkIndexPart<TLink>*)indexMemory.Pointer;
43     _header = (LinksHeader<TLink>*)indexMemory.Pointer;
44     InternalSourcesTreeMethods = _createInternalSourceTreeMethods();
45     ExternalSourcesTreeMethods = _createExternalSourceTreeMethods();
46     InternalTargetsTreeMethods = _createInternalTargetTreeMethods();
47     ExternalTargetsTreeMethods = _createExternalTargetTreeMethods();
48     UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_linksDataParts, _header);
49 }
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override void ResetPointers()
53 {
54     base.ResetPointers();
55     _linksDataParts = null;
56     _linksIndexParts = null;
57     _header = null;
58 }
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected override ref LinksHeader<TLink> GetHeaderReference() => ref *_header;
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected override ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex)
    ↳ => ref _linksDataParts[linkIndex];
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 protected override ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
    ↳ linkIndex) => ref _linksIndexParts[linkIndex];
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected override bool AreEqual(ulong first, ulong second) => first == second;
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 protected override bool LessThan(ulong first, ulong second) => first < second;
74
75 [MethodImpl(MethodImplOptions.AggressiveInlining)]
76 protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
77
78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 protected override bool GreaterThan(ulong first, ulong second) => first > second;
80
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 protected override ulong GetZero() => 0UL;
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 protected override ulong GetOne() => 1UL;

```

```

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

```

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

```

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

```

1.57 ./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 }

```

```

21
22     protected readonly TLink Break;
23     protected readonly TLink Continue;
24     protected readonly byte* Links;
25     protected readonly byte* Header;
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     protected LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
29     ↪ byte* header)
30     {
31         Links = links;
32         Header = header;
33         Break = constants.Break;
34         Continue = constants.Continue;
35     }
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected abstract TLink GetTreeRoot();
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected abstract TLink GetBasePartValue(TLink link);
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
45     ↪ rootSource, TLink rootTarget);
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
49     ↪ rootSource, TLink rootTarget);
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
53     ↪ AsRef<LinksHeader<TLink>>(Header);
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
57     ↪ AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
58     ↪ _addressToInt64Converter.Convert(link)));
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
62     {
63         ref var link = ref GetLinkReference(linkIndex);
64         return new Link<TLink>(linkIndex, link.Source, link.Target);
65     }
66
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
69     {
70         ref var firstLink = ref GetLinkReference(first);
71         ref var secondLink = ref GetLinkReference(second);
72         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
73         ↪ secondLink.Source, secondLink.Target);
74     }
75
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
78     {
79         ref var firstLink = ref GetLinkReference(first);
80         ref var secondLink = ref GetLinkReference(second);
81         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
82         ↪ secondLink.Source, secondLink.Target);
83     }
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     protected virtual TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5,
87     ↪ -5);
88
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     protected virtual void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
91     ↪ Bit<TLink>.PartialWrite(storedValue, size, 5, -5);
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     protected virtual bool GetLeftIsChildValue(TLink value)
95     {
96         unchecked
97         {
98             return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 4, 1));
99             //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
100         }
101     }

```

```

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,
100             ↪ _boolToAddressConverter.Convert(value), 4, 1);
101         storedValue = modified;
102     }
103 }
104
105 [MethodImpl(MethodImplOptions.AggressiveInlining)]
106 protected virtual bool GetRightIsChildValue(TLink value)
107 {
108     unchecked
109     {
110         return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 3, 1));
111         //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
112     }
113 }
114
115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 protected virtual void SetRightIsChildValue(ref TLink storedValue, bool value)
117 {
118     unchecked
119     {
120         var previousValue = storedValue;
121         var modified = Bit<TLink>.PartialWrite(previousValue,
122             ↪ _boolToAddressConverter.Convert(value), 3, 1);
123         storedValue = modified;
124     }
125 }
126
127 [MethodImpl(MethodImplOptions.AggressiveInlining)]
128 protected bool IsChild(TLink parent, TLink possibleChild)
129 {
130     var parentSize = GetSize(parent);
131     var childSize = GetSizeOrZero(possibleChild);
132     return GreaterThanZero(childSize) && LessOrEqualThan(childSize, parentSize);
133 }
134
135 [MethodImpl(MethodImplOptions.AggressiveInlining)]
136 protected virtual sbyte GetBalanceValue(TLink storedValue)
137 {
138     unchecked
139     {
140         var value = _addressToInt32Converter.Convert(Bit<TLink>.PartialRead(storedValue,
141             ↪ 0, 3));
142         value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
143             ↪ end of sbyte
144         return (sbyte)value;
145     }
146 }
147
148 [MethodImpl(MethodImplOptions.AggressiveInlining)]
149 protected virtual void SetBalanceValue(ref TLink storedValue, sbyte value)
150 {
151     unchecked
152     {
153         var packagedValue = _int32ToAddressConverter.Convert((byte)value >> 5 & 4 |
154             ↪ value & 3);
155         var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
156         storedValue = modified;
157     }
158 }
159
160 public TLink this[TLink index]
161 {
162     [MethodImpl(MethodImplOptions.AggressiveInlining)]
163     get
164     {
165         var root = GetTreeRoot();
166         if (GreaterOrEqualThan(index, GetSize(root)))
167         {
168             return Zero;
169         }
170     }
171 }

```



```

164     }
165     while (!EqualToZero(root))
166     {
167         var left = GetLeftOrDefault(root);
168         var leftSize = GetSizeOrZero(left);
169         if (LessThan(index, leftSize))
170         {
171             root = left;
172             continue;
173         }
174         if (AreEqual(index, leftSize))
175         {
176             return root;
177         }
178         root = GetRightOrDefault(root);
179         index = Subtract(index, Increment(leftSize));
180     }
181     return Zero; // TODO: Impossible situation exception (only if tree structure
182                 ↪ broken)
183 }
184
185 /// <summary>
186 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
187 ↪ (концом).
188 /// </summary>
189 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
190 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
191 /// <returns>Индекс искомой связи.</returns>
192 [MethodImpl(MethodImplOptions.AggressiveInlining)]
193 public TLink Search(TLink source, TLink target)
194 {
195     var root = GetTreeRoot();
196     while (!EqualToZero(root))
197     {
198         ref var rootLink = ref GetLinkReference(root);
199         var rootSource = rootLink.Source;
200         var rootTarget = rootLink.Target;
201         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
202             ↪ node.Key < root.Key
203         {
204             root = GetLeftOrDefault(root);
205         }
206         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
207             ↪ node.Key > root.Key
208         {
209             root = GetRightOrDefault(root);
210         }
211         else // node.Key == root.Key
212         {
213             return root;
214         }
215     }
216     return Zero;
217 }
218
219 // TODO: Return indices range instead of references count
220 [MethodImpl(MethodImplOptions.AggressiveInlining)]
221 public TLink CountUsages(TLink link)
222 {
223     var root = GetTreeRoot();
224     var total = GetSize(root);
225     var totalRightIgnore = Zero;
226     while (!EqualToZero(root))
227     {
228         var @base = GetBasePartValue(root);
229         if (LessOrEqualThan(@base, link))
230         {
231             root = GetRightOrDefault(root);
232         }
233         else
234         {
235             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
236             root = GetLeftOrDefault(root);
237         }
238     }
239     root = GetTreeRoot();
240     var totalLeftIgnore = Zero;

```

```

238     while (!EqualToZero(root))
239     {
240         var @base = GetBasePartValue(root);
241         if (GreaterOrEqualThan(@base, link))
242         {
243             root = GetLeftOrDefault(root);
244         }
245         else
246         {
247             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
248             root = GetRightOrDefault(root);
249         }
250     }
251     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
252 }
253
254 [MethodImpl(MethodImplOptions.AggressiveInlining)]
255 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
256 {
257     var root = GetTreeRoot();
258     if (EqualToZero(root))
259     {
260         return Continue;
261     }
262     TLink first = Zero, current = root;
263     while (!EqualToZero(current))
264     {
265         var @base = GetBasePartValue(current);
266         if (GreaterOrEqualThan(@base, link))
267         {
268             if (AreEqual(@base, link))
269             {
270                 first = current;
271             }
272             current = GetLeftOrDefault(current);
273         }
274         else
275         {
276             current = GetRightOrDefault(current);
277         }
278     }
279     if (!EqualToZero(first))
280     {
281         current = first;
282         while (true)
283         {
284             if (AreEqual(handler(GetLinkValues(current)), Break))
285             {
286                 return Break;
287             }
288             current = GetNext(current);
289             if (EqualToZero(current) || !AreEqual(GetBasePartValue(current), link))
290             {
291                 break;
292             }
293         }
294     }
295     return Continue;
296 }
297
298 [MethodImpl(MethodImplOptions.AggressiveInlining)]
299 protected override void PrintNodeValue(TLink node, StringBuilder sb)
300 {
301     ref var link = ref GetLinkReference(node);
302     sb.Append(' ');
303     sb.Append(link.Source);
304     sb.Append(' - ');
305     sb.Append(' > ');
306     sb.Append(link.Target);
307 }
308 }
309 }
310 }

```

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

```

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

```

```

5 using Platform.Collections.Methods.Trees;
6 using Platform.Converters;
7 using static System.Runtime.CompilerServices.Unsafe;
8
9 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Memory.United.Generic
12 {
13     public unsafe abstract class LinksSizeBalancedTreeMethodsBase<TLink> :
14         ↳ SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
15     {
16         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
17             ↳ UncheckedConverter<TLink, long>.Default;
18
19         protected readonly TLink Break;
20         protected readonly TLink Continue;
21         protected readonly byte* Links;
22         protected readonly byte* Header;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
26             ↳ byte* header)
27         {
28             Links = links;
29             Header = header;
30             Break = constants.Break;
31             Continue = constants.Continue;
32         }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected abstract TLink GetTreeRoot();
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected abstract TLink GetBasePartValue(TLink link);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
42             ↳ rootSource, TLink rootTarget);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
46             ↳ rootSource, TLink rootTarget);
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
50             ↳ AsRef<LinksHeader<TLink>>(Header);
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
54             ↳ AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
55             ↳ _addressToInt64Converter.Convert(link)));
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
59         {
60             ref var link = ref GetLinkReference(linkIndex);
61             return new Link<TLink>(linkIndex, link.Source, link.Target);
62         }
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
66         {
67             ref var firstLink = ref GetLinkReference(first);
68             ref var secondLink = ref GetLinkReference(second);
69             return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
70             ↳ secondLink.Source, secondLink.Target);
71         }
72
73         [MethodImpl(MethodImplOptions.AggressiveInlining)]
74         protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
75         {
76             ref var firstLink = ref GetLinkReference(first);
77             ref var secondLink = ref GetLinkReference(second);
78             return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
79             ↳ secondLink.Source, secondLink.Target);
80         }
81
82         public TLink this[TLink index]
83         {
84

```

```

74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     get
76     {
77         var root = GetTreeRoot();
78         if (GreaterOrEqualThan(index, GetSize(root)))
79         {
80             return Zero;
81         }
82         while (!EqualToZero(root))
83         {
84             var left = GetLeftOrDefault(root);
85             var leftSize = GetSizeOrZero(left);
86             if (LessThan(index, leftSize))
87             {
88                 root = left;
89                 continue;
90             }
91             if (AreEqual(index, leftSize))
92             {
93                 return root;
94             }
95             root = GetRightOrDefault(root);
96             index = Subtract(index, Increment(leftSize));
97         }
98         return Zero; // TODO: Impossible situation exception (only if tree structure
99             ↳ broken)
100     }
101
102     /// <summary>
103     /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
104     /// ↳ (концом).
105     /// </summary>
106     /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
107     /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
108     /// <returns>Индекс искомой связи.</returns>
109     [MethodImpl(MethodImplOptions.AggressiveInlining)]
110     public TLink Search(TLink source, TLink target)
111     {
112         var root = GetTreeRoot();
113         while (!EqualToZero(root))
114         {
115             ref var rootLink = ref GetLinkReference(root);
116             var rootSource = rootLink.Source;
117             var rootTarget = rootLink.Target;
118             if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
119                 ↳ node.Key < root.Key
120             {
121                 root = GetLeftOrDefault(root);
122             }
123             else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
124                 ↳ node.Key > root.Key
125             {
126                 root = GetRightOrDefault(root);
127             }
128             else // node.Key == root.Key
129             {
130                 return root;
131             }
132         }
133         return Zero;
134     }
135
136     /// TODO: Return indices range instead of references count
137     [MethodImpl(MethodImplOptions.AggressiveInlining)]
138     public TLink CountUsages(TLink link)
139     {
140         var root = GetTreeRoot();
141         var total = GetSize(root);
142         var totalRightIgnore = Zero;
143         while (!EqualToZero(root))
144         {
145             var @base = GetBasePartValue(root);
146             if (LessOrEqualThan(@base, link))
147             {
148                 root = GetRightOrDefault(root);
149             }
150             else

```

```

148         {
149             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
150             root = GetLeftOrDefault(root);
151         }
152     }
153     root = GetTreeRoot();
154     var totalLeftIgnore = Zero;
155     while (!EqualToZero(root))
156     {
157         var @base = GetBasePartValue(root);
158         if (GreaterOrEqualThan(@base, link))
159         {
160             root = GetLeftOrDefault(root);
161         }
162         else
163         {
164             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
165             root = GetRightOrDefault(root);
166         }
167     }
168     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
169 }
170
171 [MethodImpl(MethodImplOptions.AggressiveInlining)]
172 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
173     ↳ EachUsageCore(@base, GetTreeRoot(), handler);
174
175 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
176 ↳ low-level MSIL stack.
177 [MethodImpl(MethodImplOptions.AggressiveInlining)]
178 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
179 {
180     var @continue = Continue;
181     if (EqualToZero(link))
182     {
183         return @continue;
184     }
185     var linkBasePart = GetBasePartValue(link);
186     var @break = Break;
187     if (GreaterThan(linkBasePart, @base))
188     {
189         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
190         {
191             return @break;
192         }
193     }
194     else if (LessThan(linkBasePart, @base))
195     {
196         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
197         {
198             return @break;
199         }
200     }
201     else //if (linkBasePart == @base)
202     {
203         if (AreEqual(handler(GetLinkValues(link)), @break))
204         {
205             return @break;
206         }
207         if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
208         {
209             return @break;
210         }
211         if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
212         {
213             return @break;
214         }
215     }
216     return @continue;
217 }
218
219 [MethodImpl(MethodImplOptions.AggressiveInlining)]
220 protected override void PrintNodeValue(TLink node, StringBuilder sb)
221 {
222     ref var link = ref GetLinkReference(node);
223     sb.Append(' ');
224     sb.Append(link.Source);
225     sb.Append('-');
226     sb.Append('>');

```

```

225         sb.Append(link.Target);
226     }
227 }
228 }

```

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

```

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

```

```

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

```

1.60 ./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> :
            ↪ LinksSizeBalancedTreeMethodsBase<TLink>
8         {
9             [MethodImpl(MethodImplOptions.AggressiveInlining)]
10            public LinksSourcesSizeBalancedTreeMethods(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).LeftAsSource;
14
15            [MethodImpl(MethodImplOptions.AggressiveInlining)]
16            protected override ref TLink GetRightReference(TLink node) => ref
                ↪ 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) =>
                ↪ GetLinkReference(node).LeftAsSource = left;
26
27            [MethodImpl(MethodImplOptions.AggressiveInlining)]
28            protected override void SetRight(TLink node, TLink right) =>
                ↪ GetLinkReference(node).RightAsSource = right;
29
30            [MethodImpl(MethodImplOptions.AggressiveInlining)]
31            protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsSource;
32
33            [MethodImpl(MethodImplOptions.AggressiveInlining)]
34            protected override void SetSize(TLink node, TLink size) =>
                ↪ GetLinkReference(node).SizeAsSource = size;
35
36            [MethodImpl(MethodImplOptions.AggressiveInlining)]
37            protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
38
39            [MethodImpl(MethodImplOptions.AggressiveInlining)]
40            protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
41
42            [MethodImpl(MethodImplOptions.AggressiveInlining)]
43            protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
                ↪ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
                ↪ (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
44
45            [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

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

```

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

```



```

50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
    ↳ GetLinkReference(node).SizeAsTarget, value);
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
    ↳ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
    ↳ (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
    ↳ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
    ↳ (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 protected override void ClearNode(TLink node)
68 {
69     ref var link = ref GetLinkReference(node);
70     link.LeftAsTarget = Zero;
71     link.RightAsTarget = Zero;
72     link.SizeAsTarget = Zero;
73 }
74 }
75 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Memory.United.Generic
6 {
7     public unsafe class LinksTargetsSizeBalancedTreeMethods<TLink> :
8     ↳ LinksSizeBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
12             ↳ byte* header) : base(constants, links, header) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref TLink GetLeftReference(TLink node) => ref
16             ↳ GetLinkReference(node).LeftAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref TLink GetRightReference(TLink node) => ref
20             ↳ GetLinkReference(node).RightAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(TLink node, TLink left) =>
30             ↳ GetLinkReference(node).LeftAsTarget = left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(TLink node, TLink right) =>
34             ↳ GetLinkReference(node).RightAsTarget = right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsTarget;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(TLink node, TLink size) =>
41             ↳ GetLinkReference(node).SizeAsTarget = size;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

40     protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected override bool FirstIsToLeftOfSecond(TLink firstSource, TLink firstTarget,
44         ↪ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
45         ↪ (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
49         ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
50         ↪ (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     protected override void ClearNode(TLink node)
54     {
55         ref var link = ref GetLinkReference(node);
56         link.LeftAsTarget = Zero;
57         link.RightAsTarget = Zero;
58         link.SizeAsTarget = Zero;
59     }
60 }

```

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

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using static System.Runtime.CompilerServices.Unsafe;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Memory.United.Generic
10 {
11     public unsafe class UnitedMemoryLinks<TLink> : UnitedMemoryLinksBase<TLink>
12     {
13         private readonly Func<ILinksTreeMethods<TLink>> _createSourceTreeMethods;
14         private readonly Func<ILinksTreeMethods<TLink>> _createTargetTreeMethods;
15         private byte* _header;
16         private byte* _links;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
20
21         /// <summary>
22         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
23         ↪ минимальным шагом расширения базы данных.
24         /// </summary>
25         /// <param name="address">Полный путь к файлу базы данных.</param>
26         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
27         ↪ байтах.</param>
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
30             ↪ FileMappedResizableDirectMemory(address, memoryReservationStep),
31             ↪ memoryReservationStep) { }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
35             ↪ DefaultLinksSizeStep) { }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep) :
39             ↪ this(memory, memoryReservationStep, Default<LinksConstants<TLink>>.Instance,
40             ↪ IndexTreeType.Default) { }
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         public UnitedMemoryLinks(IResizableDirectMemory memory, long memoryReservationStep,
44             ↪ LinksConstants<TLink> constants, IndexTreeType indexTreeType) : base(memory,
45             ↪ memoryReservationStep, constants)
46         {
47             if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
48             {
49                 _createSourceTreeMethods = () => new
50                     ↪ LinksSourcesAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
51                 _createTargetTreeMethods = () => new
52                     ↪ LinksTargetsAvlBalancedTreeMethods<TLink>(Constants, _links, _header);
53             }
54             else
55             {

```

```

45         _createSourceTreeMethods = () => new
46         ↪ LinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
47     _createTargetTreeMethods = () => new
48     ↪ LinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
49 }
50 Init(memory, memoryReservationStep);
51 }
52 [MethodImpl(MethodImplOptions.AggressiveInlining)]
53 protected override void SetPointers(IResizableDirectMemory memory)
54 {
55     _links = (byte*)memory.Pointer;
56     _header = _links;
57     SourcesTreeMethods = _createSourceTreeMethods();
58     TargetsTreeMethods = _createTargetTreeMethods();
59     UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_links, _header);
60 }
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected override void ResetPointers()
63 {
64     base.ResetPointers();
65     _links = null;
66     _header = null;
67 }
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 protected override ref LinksHeader<TLink> GetHeaderReference() => ref
70     ↪ AsRef<LinksHeader<TLink>>(_header);
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 protected override ref RawLink<TLink> GetLinkReference(TLink linkIndex) => ref
73     ↪ AsRef<RawLink<TLink>>(_links + (LinkSizeInBytes * Convert.ToInt64(linkIndex)));
74 }
75 }

```

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

```

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

```

```

41 // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
42   ↳ нужно использовать не список а дерево, так как так можно быстрее проверить на
43   ↳ наличие связи внутри
44 protected ILinksListMethods<TLink> UnusedLinksListMethods;
45
46 /// <summary>
47 /// Возвращает общее число связей находящихся в хранилище.
48 /// </summary>
49 protected virtual TLink Total
50 {
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     get
53     {
54         ref var header = ref GetHeaderReference();
55         return Subtract(header.AllocatedLinks, header.FreeLinks);
56     }
57 }
58
59 public virtual LinksConstants<TLink> Constants
60 {
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     get;
63 }
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
67   ↳ memoryReservationStep, LinksConstants<TLink> constants)
68 {
69     _memory = memory;
70     _memoryReservationStep = memoryReservationStep;
71     Constants = constants;
72 }
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
76   ↳ memoryReservationStep) : this(memory, memoryReservationStep,
77   ↳ Default<LinksConstants<TLink>>.Instance) { }
78
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
81 {
82     if (memory.ReservedCapacity < memoryReservationStep)
83     {
84         memory.ReservedCapacity = memoryReservationStep;
85     }
86     SetPointers(memory);
87     ref var header = ref GetHeaderReference();
88     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
89     memory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) * LinkSizeInBytes) +
90       ↳ LinkHeaderSizeInBytes;
91     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
92     header.ReservedLinks = ConvertToAddress((memory.ReservedCapacity -
93       ↳ LinkHeaderSizeInBytes) / LinkSizeInBytes);
94 }
95
96 [MethodImpl(MethodImplOptions.AggressiveInlining)]
97 public virtual TLink Count(IList<TLink> restrictions)
98 {
99     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
100     if (restrictions.Count == 0)
101     {
102         return Total;
103     }
104     var constants = Constants;
105     var any = constants.Any;
106     var index = restrictions[constants.IndexPart];
107     if (restrictions.Count == 1)
108     {
109         if (AreEqual(index, any))
110         {
111             return Total;
112         }
113         return Exists(index) ? GetOne() : GetZero();
114     }
115     if (restrictions.Count == 2)
116     {
117         var value = restrictions[1];
118         if (AreEqual(index, any))
119         {
120

```

```

113         if (AreEqual(value, any))
114         {
115             return Total; // Any - как отсутствие ограничения
116         }
117         return Add(SourcesTreeMethods.CountUsages(value),
118             ↪ TargetsTreeMethods.CountUsages(value));
119     }
120     else
121     {
122         if (!Exists(index))
123         {
124             return GetZero();
125         }
126         if (AreEqual(value, any))
127         {
128             return GetOne();
129         }
130         ref var storedLinkValue = ref GetLinkReference(index);
131         if (AreEqual(storedLinkValue.Source, value) ||
132             ↪ AreEqual(storedLinkValue.Target, value))
133         {
134             return GetOne();
135         }
136         return GetZero();
137     }
138 }
139 if (restrictions.Count == 3)
140 {
141     var source = restrictions[constants.SourcePart];
142     var target = restrictions[constants.TargetPart];
143     if (AreEqual(index, any))
144     {
145         if (AreEqual(source, any) && AreEqual(target, any))
146         {
147             return Total;
148         }
149         else if (AreEqual(source, any))
150         {
151             return TargetsTreeMethods.CountUsages(target);
152         }
153         else if (AreEqual(target, any))
154         {
155             return SourcesTreeMethods.CountUsages(source);
156         }
157         else //if(source != Any && target != Any)
158         {
159             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
160             var link = SourcesTreeMethods.Search(source, target);
161             return AreEqual(link, constants.Null) ? GetZero() : GetOne();
162         }
163     }
164     else
165     {
166         if (!Exists(index))
167         {
168             return GetZero();
169         }
170         if (AreEqual(source, any) && AreEqual(target, any))
171         {
172             return GetOne();
173         }
174         ref var storedLinkValue = ref GetLinkReference(index);
175         if (!AreEqual(source, any) && !AreEqual(target, any))
176         {
177             if (AreEqual(storedLinkValue.Source, source) &&
178                 ↪ AreEqual(storedLinkValue.Target, target))
179             {
180                 return GetOne();
181             }
182             return GetZero();
183         }
184         var value = default(TLink);
185         if (AreEqual(source, any))
186         {
187             value = target;
188         }
189         if (AreEqual(target, any))
190         {
191             value = source;
192         }
193         return value;
194     }
195 }

```

```

188         value = source;
189     }
190     if (AreEqual(storedLinkValue.Source, value) ||
        ↪ AreEqual(storedLinkValue.Target, value))
191     {
192         return GetOne();
193     }
194     return GetZero();
195 }
196 }
197 throw new NotSupportedException("Другие размеры и способы ограничений не
        ↪ поддерживаются.");
198 }
199
200 [MethodImpl(MethodImplOptions.AggressiveInlining)]
201 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
202 {
203     var constants = Constants;
204     var @break = constants.Break;
205     if (restrictions.Count == 0)
206     {
207         for (var link = GetOne(); LessOrEqualThan(link,
            ↪ GetHeaderReference().AllocatedLinks); link = Increment(link))
208         {
209             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
210             {
211                 return @break;
212             }
213         }
214         return @break;
215     }
216     var @continue = constants.Continue;
217     var any = constants.Any;
218     var index = restrictions[constants.IndexPart];
219     if (restrictions.Count == 1)
220     {
221         if (AreEqual(index, any))
222         {
223             return Each(handler, Array.Empty<TLink>());
224         }
225         if (!Exists(index))
226         {
227             return @continue;
228         }
229         return handler(GetLinkStruct(index));
230     }
231     if (restrictions.Count == 2)
232     {
233         var value = restrictions[1];
234         if (AreEqual(index, any))
235         {
236             if (AreEqual(value, any))
237             {
238                 return Each(handler, Array.Empty<TLink>());
239             }
240             if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
241             {
242                 return @break;
243             }
244             return Each(handler, new Link<TLink>(index, any, value));
245         }
246         else
247         {
248             if (!Exists(index))
249             {
250                 return @continue;
251             }
252             if (AreEqual(value, any))
253             {
254                 return handler(GetLinkStruct(index));
255             }
256             ref var storedLinkValue = ref GetLinkReference(index);
257             if (AreEqual(storedLinkValue.Source, value) ||
                AreEqual(storedLinkValue.Target, value))
258             {
259                 return handler(GetLinkStruct(index));
260             }
261             return @continue;
262         }
263     }

```

```

263     }
264 }
265 if (restrictions.Count == 3)
266 {
267     var source = restrictions[constants.SourcePart];
268     var target = restrictions[constants.TargetPart];
269     if (AreEqual(index, any))
270     {
271         if (AreEqual(source, any) && AreEqual(target, any))
272         {
273             return Each(handler, Array.Empty<TLink>());
274         }
275         else if (AreEqual(source, any))
276         {
277             return TargetsTreeMethods.EachUsage(target, handler);
278         }
279         else if (AreEqual(target, any))
280         {
281             return SourcesTreeMethods.EachUsage(source, handler);
282         }
283         else //if(source != Any && target != Any)
284         {
285             var link = SourcesTreeMethods.Search(source, target);
286             return AreEqual(link, constants.Null) ? @continue :
287                 ↪ handler(GetLinkStruct(link));
288         }
289     }
290     else
291     {
292         if (!Exists(index))
293         {
294             return @continue;
295         }
296         if (AreEqual(source, any) && AreEqual(target, any))
297         {
298             return handler(GetLinkStruct(index));
299         }
300         ref var storedLinkValue = ref GetLinkReference(index);
301         if (!AreEqual(source, any) && !AreEqual(target, any))
302         {
303             if (AreEqual(storedLinkValue.Source, source) &&
304                 AreEqual(storedLinkValue.Target, target))
305             {
306                 return handler(GetLinkStruct(index));
307             }
308             return @continue;
309         }
310         var value = default(TLink);
311         if (AreEqual(source, any))
312         {
313             value = target;
314         }
315         if (AreEqual(target, any))
316         {
317             value = source;
318         }
319         if (AreEqual(storedLinkValue.Source, value) ||
320             AreEqual(storedLinkValue.Target, value))
321         {
322             return handler(GetLinkStruct(index));
323         }
324         return @continue;
325     }
326 }
327 throw new NotSupportedException("Другие размеры и способы ограничений не
328     ↪ поддерживаются.");
329
330 /// <remarks>
331 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
332     ↪ в другом месте (но не в менеджере памяти, а в логике Links)
333 /// </remarks>
334 [MethodImpl(MethodImplOptions.AggressiveInlining)]
335 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
336 {
337     var constants = Constants;
338     var @null = constants.Null;
339     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
343     ↪ предварительно заполнено нулями
344     if (!AreEqual(link.Source, @null))
345     {
346         SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
347     }
348     if (!AreEqual(link.Target, @null))
349     {
350         TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
351     }
352     link.Source = substitution[constants.SourcePart];
353     link.Target = substitution[constants.TargetPart];
354     if (!AreEqual(link.Source, @null))
355     {
356         SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
357     }
358     if (!AreEqual(link.Target, @null))
359     {
360         TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
361     }
362     return linkIndex;
363 }
364
365 /// <remarks>
366 /// TODO0: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
367 ↪ пространство
368 /// </remarks>
369 [MethodImpl(MethodImplOptions.AggressiveInlining)]
370 public virtual TLink Create(ICollection<TLink> restrictions)
371 {
372     ref var header = ref GetHeaderReference();
373     var freeLink = header.FirstFreeLink;
374     if (!AreEqual(freeLink, Constants.Null))
375     {
376         UnusedLinksListMethods.Detach(freeLink);
377     }
378     else
379     {
380         var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
381         if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
382         {
383             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
384         }
385         if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks))
386         {
387             _memory.ReservedCapacity += _memory.ReservationStep;
388             SetPointers(_memory);
389             header = ref GetHeaderReference();
390             header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
391             ↪ LinkSizeInBytes);
392         }
393         header.AllocatedLinks = Increment(header.AllocatedLinks);
394         _memory.UsedCapacity += LinkSizeInBytes;
395         freeLink = header.AllocatedLinks;
396     }
397     return freeLink;
398 }
399
400 [MethodImpl(MethodImplOptions.AggressiveInlining)]
401 public virtual void Delete(ICollection<TLink> restrictions)
402 {
403     ref var header = ref GetHeaderReference();
404     var link = restrictions[Constants.IndexPart];
405     if (LessThan(link, header.AllocatedLinks)
406     {
407         UnusedLinksListMethods.AttachAsFirst(link);
408     }
409     else if (AreEqual(link, header.AllocatedLinks)
410     {
411         header.AllocatedLinks = Decrement(header.AllocatedLinks);
412         _memory.UsedCapacity -= LinkSizeInBytes;
413         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
414         ↪ пока не дойдём до первой существующей связи
415         // 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     [MethodImpl(MethodImplOptions.AggressiveInlining)]
422     public IList<TLink> GetLinkStruct(TLink linkIndex)
423     {
424         ref var link = ref GetLinkReference(linkIndex);
425         return new Link<TLink>(linkIndex, link.Source, link.Target);
426     }
427
428     /// <remarks>
429     /// TODO0: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
430     ↪ адрес реально поменялся
431     ///
432     /// Указатель this.links может быть в том же месте,
433     /// так как 0-я связь не используется и имеет такой же размер как Header,
434     /// поэтому header размещается в том же месте, что и 0-я связь
435     /// </remarks>
436     [MethodImpl(MethodImplOptions.AggressiveInlining)]
437     protected abstract void SetPointers(IResizableDirectMemory memory);
438
439     [MethodImpl(MethodImplOptions.AggressiveInlining)]
440     protected virtual void ResetPointers()
441     {
442         SourcesTreeMethods = null;
443         TargetsTreeMethods = null;
444         UnusedLinksListMethods = null;
445     }
446
447     [MethodImpl(MethodImplOptions.AggressiveInlining)]
448     protected abstract ref LinksHeader<TLink> GetHeaderReference();
449
450     [MethodImpl(MethodImplOptions.AggressiveInlining)]
451     protected abstract ref RawLink<TLink> GetLinkReference(TLink linkIndex);
452
453     [MethodImpl(MethodImplOptions.AggressiveInlining)]
454     protected virtual bool Exists(TLink link)
455     => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
456     && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
457     && !IsUnusedLink(link);
458
459     [MethodImpl(MethodImplOptions.AggressiveInlining)]
460     protected virtual bool IsUnusedLink(TLink linkIndex)
461     {
462         if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
463         ↪ is not needed
464         {
465             ref var link = ref GetLinkReference(linkIndex);
466             return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
467         }
468         else
469         {
470             return true;
471         }
472     }
473
474     [MethodImpl(MethodImplOptions.AggressiveInlining)]
475     protected virtual TLink GetOne() => _one;
476
477     [MethodImpl(MethodImplOptions.AggressiveInlining)]
478     protected virtual TLink GetZero() => default;
479
480     [MethodImpl(MethodImplOptions.AggressiveInlining)]
481     protected virtual bool AreEqual(TLink first, TLink second) =>
482     ↪ _equalityComparer.Equals(first, second);
483
484     [MethodImpl(MethodImplOptions.AggressiveInlining)]
485     protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
486     ↪ second) < 0;
487
488     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Methods.Lists;
3  using Platform.Converters;
4  using static System.Runtime.CompilerServices.Unsafe;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Memory.United.Generic
9  {
10     public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
11         ↪ ILinksListMethods<TLink>
12     {
13         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
14             ↪ UncheckedConverter<TLink, long>.Default;
15
16         private readonly byte* _links;
17         private readonly byte* _header;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnusedLinksListMethods(byte* links, byte* header)
21         {
22             _links = links;
23             _header = header;
24         }
25     }
26 }

```

```

23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
25     ↪ AsRef<LinksHeader<TLink>>(_header);
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
29     ↪ AsRef<RawLink<TLink>>(_links + (RawLink<TLink>.SizeInBytes *
30     ↪ _addressToInt64Converter.Convert(link)));
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override TLink GetPrevious(TLink element) => GetLinkReference(element).Source;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override TLink GetNext(TLink element) => GetLinkReference(element).Target;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override TLink GetSize() => GetHeaderReference().FreeLinks;
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
49     ↪ element;
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
53     ↪ element;
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override void SetPrevious(TLink element, TLink previous) =>
57     ↪ GetLinkReference(element).Source = previous;
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override void SetNext(TLink element, TLink next) =>
61     ↪ GetLinkReference(element).Target = next;
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
65 }

```

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

```

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

```

```

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 }

```

1.67 ./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 :
    ↪ LinksSizeBalancedTreeMethodsBase<uint>
9      {
10         protected new readonly RawLink<uint>* Links;
11         protected new readonly LinksHeader<uint>* Header;
12
13         protected UInt32LinksSizeBalancedTreeMethodsBase(LinksConstants<uint> constants,
    ↪ RawLink<uint>* links, LinksHeader<uint>* header)
14             : base(constants, (byte*)links, (byte*)header)
15         {
16             Links = links;
17             Header = header;
18         }
19
20         [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 FirstIsToLeftOfSecond(uint first, uint second)
67     {
68         ref var firstLink = ref Links[first];
69         ref var secondLink = ref Links[second];
70         return FirstIsToLeftOfSecond(firstLink.Source, firstLink.Target,
71             ↪ secondLink.Source, secondLink.Target);
72     }
73
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     protected override bool FirstIsToTheRightOfSecond(uint first, uint second)
76     {
77         ref var firstLink = ref Links[first];
78         ref var secondLink = ref Links[second];
79         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
80             ↪ secondLink.Source, secondLink.Target);
81     }
82
83     [MethodImpl(MethodImplOptions.AggressiveInlining)]
84     protected override ref LinksHeader<uint> GetHeaderReference() => ref *Header;
85
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     protected override ref RawLink<uint> GetLinkReference(uint link) => ref Links[link];
88 }

```

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

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.United.Specific
6  {
7      public unsafe class UInt32LinksSourcesSizeBalancedTreeMethods :
8          ↪ UInt32LinksSizeBalancedTreeMethodsBase
9      {
10         public UInt32LinksSourcesSizeBalancedTreeMethods(LinksConstants<uint> constants,
11             ↪ RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links, header) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsSource;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected override ref uint GetRightReference(uint node) => ref
18             ↪ Links[node].RightAsSource;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override uint GetLeft(uint node) => Links[node].LeftAsSource;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override uint GetRight(uint node) => Links[node].RightAsSource;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override void SetLeft(uint node, uint left) => Links[node].LeftAsSource = left;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override void SetRight(uint node, uint right) => Links[node].RightAsSource =
31             ↪ right;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override uint GetSize(uint node) => Links[node].SizeAsSource;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override void SetSize(uint node, uint size) => Links[node].SizeAsSource = size;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override uint GetTreeRoot() => Header->RootAsSource;
41     }

```

```

38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override uint GetBasePartValue(uint link) => Links[link].Source;
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override bool FirstIsToLeftOfSecond(uint firstSource, uint firstTarget,
43     ↪ uint secondSource, uint secondTarget)
44     => firstSource < secondSource || (firstSource == secondSource && firstTarget <
45     ↪ secondTarget);
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override bool FirstIsToTheRightOfSecond(uint firstSource, uint firstTarget,
49     ↪ uint secondSource, uint secondTarget)
50     => firstSource > secondSource || (firstSource == secondSource && firstTarget >
51     ↪ secondTarget);
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected override void ClearNode(uint node)
55     {
56         ref var link = ref Links[node];
57         link.LeftAsSource = 0U;
58         link.RightAsSource = 0U;
59         link.SizeAsSource = 0U;
60     }
61 }

```

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

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.United.Specific
6  {
7      public unsafe class UInt32LinksTargetsSizeBalancedTreeMethods :
8      ↪ UInt32LinksSizeBalancedTreeMethodsBase
9      {
10         public UInt32LinksTargetsSizeBalancedTreeMethods(LinksConstants<uint> constants,
11         ↪ RawLink<uint>* links, LinksHeader<uint>* header) : base(constants, links, header) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected override ref uint GetLeftReference(uint node) => ref Links[node].LeftAsTarget;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected override ref uint GetRightReference(uint node) => ref
18         ↪ Links[node].RightAsTarget;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override uint GetLeft(uint node) => Links[node].LeftAsTarget;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override uint GetRight(uint node) => Links[node].RightAsTarget;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override void SetLeft(uint node, uint left) => Links[node].LeftAsTarget = left;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected override void SetRight(uint node, uint right) => Links[node].RightAsTarget =
31         ↪ right;
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected override uint GetSize(uint node) => Links[node].SizeAsTarget;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override void SetSize(uint node, uint size) => Links[node].SizeAsTarget = size;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override uint GetTreeRoot() => Header->RootAsTarget;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected override uint GetBasePartValue(uint link) => Links[link].Target;
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override bool FirstIsToLeftOfSecond(uint firstSource, uint firstTarget,
47         ↪ uint secondSource, uint secondTarget)
48         => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
49         ↪ secondSource);
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

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

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Memory;
4  using Platform.Singletons;
5  using Platform.Data.Doublets.Memory.United.Generic;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Memory.United.Specific
10 {
11     /// <summary>
12     /// <para>Represents a low-level implementation of direct access to resizable memory, for
13     ↪ organizing the storage of links with addresses represented as <see cref="uint" />.</para>
14     /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
15     ↪ размером, для организации хранения связей с адресами представленными в виде <see
16     ↪ cref="uint"/>.</para>
17     /// </summary>
18     public unsafe class UInt32UnitedMemoryLinks : UnitedMemoryLinksBase<uint>
19     {
20         private readonly Func<ILinksTreeMethods<uint>> _createSourceTreeMethods;
21         private readonly Func<ILinksTreeMethods<uint>> _createTargetTreeMethods;
22         private LinksHeader<uint>* _header;
23         private RawLink<uint>* _links;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public UInt32UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
27
28         /// <summary>
29         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
30         ↪ минимальным шагом расширения базы данных.
31         /// </summary>
32         /// <param name="address">Полный путь к файлу базы данных.</param>
33         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
34         ↪ байтах.</param>
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public UInt32UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
37         ↪ FileMappedResizableDirectMemory(address, memoryReservationStep),
38         ↪ memoryReservationStep) { }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public UInt32UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
42         ↪ DefaultLinksSizeStep) { }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
46         ↪ memoryReservationStep) : this(memory, memoryReservationStep,
47         ↪ Default<LinksConstants<uint>>.Instance, IndexTreeType.Default) { }
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public UInt32UnitedMemoryLinks(IResizableDirectMemory memory, long
51         ↪ memoryReservationStep, LinksConstants<uint> constants, IndexTreeType indexTreeType)
52         ↪ : base(memory, memoryReservationStep, constants)
53         {
54             _createSourceTreeMethods = () => new
55             ↪ UInt32LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
56             _createTargetTreeMethods = () => new
57             ↪ UInt32LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
58             Init(memory, memoryReservationStep);
59         }
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

1.71 ./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 }

```

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

```

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

```

```

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,
72             ↪ secondLink.Source, secondLink.Target);
73     }
74
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
77     {
78         ref var firstLink = ref Links[first];
79         ref var secondLink = ref Links[second];
80         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
81             ↪ secondLink.Source, secondLink.Target);
82     }
83
84     [MethodImpl(MethodImplOptions.AggressiveInlining)]
85     protected override ulong GetSizeValue(ulong value) => (value & 4294967264UL) >> 5;
86
87     [MethodImpl(MethodImplOptions.AggressiveInlining)]
88     protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
89         ↪ storedValue & 31UL | (size & 134217727UL) << 5;
90
91     [MethodImpl(MethodImplOptions.AggressiveInlining)]
92     protected override bool GetLeftIsChildValue(ulong value) => (value & 16UL) >> 4 == 1UL;
93
94     [MethodImpl(MethodImplOptions.AggressiveInlining)]
95     protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
96         ↪ storedValue = storedValue & 4294967279UL | (As<bool, byte>(ref value) & 1UL) << 4;
97
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     protected override bool GetRightIsChildValue(ulong value) => (value & 8UL) >> 3 == 1UL;
100
101     [MethodImpl(MethodImplOptions.AggressiveInlining)]
102     protected override sbyte GetBalanceValue(ulong value) => unchecked((sbyte)(value & 7UL |
103         ↪ 0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of
104         ↪ sbyte
105
106     [MethodImpl(MethodImplOptions.AggressiveInlining)]
107     protected override void SetBalanceValue(ref ulong storedValue, sbyte value) =>
108         ↪ storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
109         ↪ value & 3) & 7UL);
110
111     [MethodImpl(MethodImplOptions.AggressiveInlining)]
112     protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
113
114     [MethodImpl(MethodImplOptions.AggressiveInlining)]
115     protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
116 }

```

1.73 ./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     }

```

```

15 {
16     Links = links;
17     Header = header;
18 }
19
20 [MethodImpl(MethodImplOptions.AggressiveInlining)]
21 protected override ulong GetZero() => 0UL;
22
23 [MethodImpl(MethodImplOptions.AggressiveInlining)]
24 protected override bool EqualToZero(ulong value) => value == 0UL;
25
26 [MethodImpl(MethodImplOptions.AggressiveInlining)]
27 protected override bool AreEqual(ulong first, ulong second) => first == second;
28
29 [MethodImpl(MethodImplOptions.AggressiveInlining)]
30 protected override bool GreaterThanZero(ulong value) => value > 0UL;
31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 protected override bool GreaterThan(ulong first, ulong second) => first > second;
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
37
38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
    ↳ always true for ulong
40
41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
    ↳ always >= 0 for ulong
43
44 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
    ↳ for ulong
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 protected override bool LessThan(ulong first, ulong second) => first < second;
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected override ulong Increment(ulong value) => ++value;
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 protected override ulong Decrement(ulong value) => --value;
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected override ulong Add(ulong first, ulong second) => first + second;
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override ulong Subtract(ulong first, ulong second) => first - second;
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override bool 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 }

```

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

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.United.Specific
6  {
7      public unsafe class UInt64LinksSourcesAvlBalancedTreeMethods :
8          ↳ UInt64LinksAvlBalancedTreeMethodsBase
9      {
10
11          public UInt64LinksSourcesAvlBalancedTreeMethods(LinksConstants<ulong> constants,
12              ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
13              ↳ { }
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          protected override ref ulong GetLeftReference(ulong node) => ref
17              ↳ Links[node].LeftAsSource;
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          protected override ref ulong GetRightReference(ulong node) => ref
21              ↳ Links[node].RightAsSource;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
31              ↳ left;
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
35              ↳ right;
36
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsSource);
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
42              ↳ Links[node].SizeAsSource, size);
43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          protected override bool GetLeftIsChild(ulong node) =>
46              ↳ GetLeftIsChildValue(Links[node].SizeAsSource);
47
48          //[MethodImpl(MethodImplOptions.AggressiveInlining)]
49          //protected override bool GetLeftIsChild(ulong node) => IsChild(node, GetLeft(node));
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          protected override void SetLeftIsChild(ulong node, bool value) =>
53              ↳ SetLeftIsChildValue(ref Links[node].SizeAsSource, value);
54
55          [MethodImpl(MethodImplOptions.AggressiveInlining)]
56          protected override bool GetRightIsChild(ulong node) =>
57              ↳ GetRightIsChildValue(Links[node].SizeAsSource);
58
59          //[MethodImpl(MethodImplOptions.AggressiveInlining)]
60          //protected override bool GetRightIsChild(ulong node) => IsChild(node, GetRight(node));
61
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          protected override void SetRightIsChild(ulong node, bool value) =>
64              ↳ SetRightIsChildValue(ref Links[node].SizeAsSource, value);
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          protected override sbyte GetBalance(ulong node) =>
68              ↳ GetBalanceValue(Links[node].SizeAsSource);
69
70          [MethodImpl(MethodImplOptions.AggressiveInlining)]
71          protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
72              ↳ Links[node].SizeAsSource, value);
73
74          [MethodImpl(MethodImplOptions.AggressiveInlining)]
75          protected override ulong GetTreeRoot() => Header->RootAsSource;
76
77          [MethodImpl(MethodImplOptions.AggressiveInlining)]
78          protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
79
80      }
81  }
```

```

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

```

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

```

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

```

```

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

```

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

```

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

```

```

50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
    ↳ Links[node].SizeAsTarget, value);
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected override ulong GetTreeRoot() => Header->RootAsTarget;
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
    ↳ ulong secondSource, ulong secondTarget)
61     => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
    ↳ secondSource);
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
    ↳ ulong secondSource, ulong secondTarget)
65     => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
    ↳ secondSource);
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 protected override void ClearNode(ulong node)
69 {
70     ref var link = ref Links[node];
71     link.LeftAsTarget = OUL;
72     link.RightAsTarget = OUL;
73     link.SizeAsTarget = OUL;
74 }
75 }
76 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Memory.United.Specific
6 {
7     public unsafe class UInt64LinksTargetsSizeBalancedTreeMethods :
8         ↳ UInt64LinksSizeBalancedTreeMethodsBase
9     {
10         public UInt64LinksTargetsSizeBalancedTreeMethods(LinksConstants<ulong> constants,
11             ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
12             ↳ { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetLeftReference(ulong node) => ref
16             ↳ Links[node].LeftAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20             ↳ Links[node].RightAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
30             ↳ left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
34             ↳ right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
41             ↳ size;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override ulong GetTreeRoot() => Header->RootAsTarget;
45     }
46 }

```

```

38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
40
41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
43     ↳ ulong secondSource, ulong secondTarget)
44     => firstTarget < secondTarget || (firstTarget == secondTarget && firstSource <
45     ↳ secondSource);
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
49     ↳ ulong secondSource, ulong secondTarget)
50     => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
51     ↳ secondSource);
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected override void ClearNode(ulong node)
55 {
56     ref var link = ref Links[node];
57     link.LeftAsTarget = OUL;
58     link.RightAsTarget = OUL;
59     link.SizeAsTarget = OUL;
60 }
61
62 }
63
64 }

```

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

```

1 using System;
2 using System.Runtime.CompilerServices;
3 using Platform.Memory;
4 using Platform.Singletons;
5 using Platform.Data.Doublets.Memory.United.Generic;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Memory.United.Specific
10 {
11     /// <summary>
12     /// <para>Represents a low-level implementation of direct access to resizable memory, for
13     ↪ organizing the storage of links with addresses represented as <see cref="ulong"
14     ↪ >/>.</para>
15     /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
16     ↪ размером, для организации хранения связей с адресами представленными в виде <see
17     ↪ cref="ulong"/>.</para>
18     /// </summary>
19     public unsafe class UInt64UnitedMemoryLinks : UnitedMemoryLinksBase<ulong>
20     {
21         private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
22         private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
23         private LinksHeader<ulong>* _header;
24         private RawLink<ulong>* _links;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public UInt64UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
28
29         /// <summary>
30         /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
31         ↪ минимальным шагом расширения базы данных.
32         /// </summary>
33         /// <param name="address">Полный путь к файлу базы данных.</param>
34         /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
35         ↪ байтах.</param>
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public UInt64UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
38         ↪ FileMappedResizableDirectMemory(address, memoryReservationStep),
39         ↪ memoryReservationStep) { }
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public UInt64UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
43         ↪ DefaultLinksSizeStep) { }
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
47         ↪ memoryReservationStep) : this(memory, memoryReservationStep,
48         ↪ Default<LinksConstants<ulong>>.Instance, IndexTreeType.Default) { }
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51

```



```

40 public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
    ↳ memoryReservationStep, LinksConstants<ulong> constants, IndexTreeType indexTreeType)
    ↳ : base(memory, memoryReservationStep, constants)
41 {
42     if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
43     {
44         _createSourceTreeMethods = () => new
            ↳ UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
45         _createTargetTreeMethods = () => new
            ↳ UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
46     }
47     else
48     {
49         _createSourceTreeMethods = () => new
            ↳ UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
50         _createTargetTreeMethods = () => new
            ↳ UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
51     }
52     Init(memory, memoryReservationStep);
53 }
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 protected override void SetPointers(IResizableDirectMemory memory)
57 {
58     _header = (LinksHeader<ulong>*)memory.Pointer;
59     _links = (RawLink<ulong>*)memory.Pointer;
60     SourcesTreeMethods = _createSourceTreeMethods();
61     TargetsTreeMethods = _createTargetTreeMethods();
62     UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
63 }
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override void ResetPointers()
67 {
68     base.ResetPointers();
69     _links = null;
70     _header = null;
71 }
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
    ↳ _links[linkIndex];
78
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 protected override bool AreEqual(ulong first, ulong second) => first == second;
81
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 protected override bool LessThan(ulong first, ulong second) => first < second;
84
85 [MethodImpl(MethodImplOptions.AggressiveInlining)]
86 protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
87
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 protected override bool GreaterThan(ulong first, ulong second) => first > second;
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 protected override ulong GetZero() => 0UL;
96
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 protected override ulong GetOne() => 1UL;
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 protected override long ConvertToInt64(ulong value) => (long)value;
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 protected override ulong ConvertToAddress(long value) => (ulong)value;
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 protected override ulong Add(ulong first, ulong second) => first + second;
108
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 protected override ulong Subtract(ulong first, ulong second) => first - second;
111

```

```

112     [MethodImpl(MethodImplOptions.AggressiveInlining)]
113     protected override ulong Increment(ulong link) => ++link;
114
115     [MethodImpl(MethodImplOptions.AggressiveInlining)]
116     protected override ulong Decrement(ulong link) => --link;
117 }
118 }

```

1.79 ./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 }

```

1.80 ./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         }
44     }
45 }

```

```

38         return target;
39     }
40 }
41 }

```

1.81 ./csharp/Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConverter.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Converters;
5  using System.Runtime.CompilerServices;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class LinkToItsFrequencyNumberConverter<TLink> : LinksOperatorBase<TLink>,
12         ⇨ IConverter<Doublet<TLink>, TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ⇨ EqualityComparer<TLink>.Default;
16
17         private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
18         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public LinkToItsFrequencyNumberConverter(
22             ILinks<TLink> links,
23             IProperty<TLink, TLink> frequencyPropertyOperator,
24             IConverter<TLink> unaryNumberToAddressConverter)
25             : base(links)
26         {
27             _frequencyPropertyOperator = frequencyPropertyOperator;
28             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public TLink Convert(Doublet<TLink> doublet)
33         {
34             var links = _links;
35             var link = links.SearchOrDefault(doublet.Source, doublet.Target);
36             if (_equalityComparer.Equals(link, default))
37             {
38                 throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
39             }
40             var frequency = _frequencyPropertyOperator.Get(link);
41             if (_equalityComparer.Equals(frequency, default))
42             {
43                 return default;
44             }
45             var frequencyNumber = links.GetSource(frequency);
46             return _unaryNumberToAddressConverter.Convert(frequencyNumber);
47         }
48     }
49 }

```

1.82 ./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 }

```

```

22     }
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     public TLink Convert(int power)
26     {
27         Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
28         ↪ - 1), nameof(power));
29         if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
30         {
31             return _unaryNumberPowersOf2[power];
32         }
33         var previousPowerOf2 = Convert(power - 1);
34         var powerOf2 = _links.GetOrCreate(previousPowerOf2, previousPowerOf2);
35         _unaryNumberPowersOf2[power] = powerOf2;
36         return powerOf2;
37     }
38 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Converters;
4  using Platform.Numbers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Numbers.Unary
9  {
10     public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>,
11     ↪ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14         ↪ EqualityComparer<TLink>.Default;
15         private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
16         ↪ UncheckedConverter<TLink, ulong>.Default;
17         private static readonly UncheckedConverter<ulong, TLink> _uint64ToAddressConverter =
18         ↪ UncheckedConverter<ulong, TLink>.Default;
19         private static readonly TLink _zero = default;
20         private static readonly TLink _one = Arithmetic.Increment(_zero);
21
22         private readonly Dictionary<TLink, TLink> _unaryToUInt64;
23         private readonly TLink _unaryOne;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
27             : base(links)
28         {
29             _unaryOne = unaryOne;
30             _unaryToUInt64 = CreateUnaryToUInt64Dictionary(links, unaryOne);
31         }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public TLink Convert(TLink unaryNumber)
35         {
36             if (_equalityComparer.Equals(unaryNumber, default))
37             {
38                 return default;
39             }
40             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
41             {
42                 return _one;
43             }
44             var links = _links;
45             var source = links.GetSource(unaryNumber);
46             var target = links.GetTarget(unaryNumber);
47             if (_equalityComparer.Equals(source, target))
48             {
49                 return _unaryToUInt64[unaryNumber];
50             }
51             else
52             {
53                 var result = _unaryToUInt64[source];
54                 TLink lastValue;
55                 while (!_unaryToUInt64.TryGetValue(target, out lastValue))
56                 {
57                     source = links.GetSource(target);
58                     result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
59                     target = links.GetTarget(target);
60                 }
61             }
62         }
63     }
64 }

```

```

57         result = Arithmetic<TLink>.Add(result, lastValue);
58         return result;
59     }
60 }
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 private static Dictionary<TLink, TLink> CreateUnaryToUInt64Dictionary(ILinks<TLink>
    ↪ links, TLink unaryOne)
64 {
65     var unaryToUInt64 = new Dictionary<TLink, TLink>
66     {
67         { unaryOne, _one }
68     };
69     var unary = unaryOne;
70     var number = _one;
71     for (var i = 1; i < 64; i++)
72     {
73         unary = links.GetOrCreate(unary, unary);
74         number = Double(number);
75         unaryToUInt64.Add(unary, number);
76     }
77     return unaryToUInt64;
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 private static TLink Double(TLink number) =>
    ↪ _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
82 }
83 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Reflection;
4  using Platform.Converters;
5  using Platform.Numbers;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↪ EqualityComparer<TLink>.Default;
14         private static readonly TLink _zero = default;
15         private static readonly TLink _one = Arithmetic.Increment(_zero);
16
17         private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
    ↪ TLink> powerOf2ToUnaryNumberConverter) : base(links) => _unaryNumberPowerOf2Indicies
    ↪ = CreateUnaryNumberPowerOf2IndiciesDictionary(powerOf2ToUnaryNumberConverter);
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public TLink Convert(TLink sourceNumber)
24         {
25             var links = _links;
26             var nullConstant = links.Constants.Null;
27             var source = sourceNumber;
28             var target = nullConstant;
29             if (!_equalityComparer.Equals(source, nullConstant))
30             {
31                 while (true)
32                 {
33                     if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
34                     {
35                         SetBit(ref target, powerOf2Index);
36                         break;
37                     }
38                     else
39                     {
40                         powerOf2Index = _unaryNumberPowerOf2Indicies[links.GetSource(source)];
41                         SetBit(ref target, powerOf2Index);
42                         source = links.GetTarget(source);
43                     }
44                 }
45             }
46         }
47     }
48 }

```

```

46         return target;
47     }
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     private static Dictionary<TLink, int>
        ↪ CreateUnaryNumberPowerOf2IndiciesDictionary(IConverter<int, TLink>
        ↪ powerOf2ToUnaryNumberConverter)
51     {
52         var unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
53         for (int i = 0; i < NumericType<TLink>.BitsSize; i++)
54         {
55             unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
56         }
57         return unaryNumberPowerOf2Indicies;
58     }
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     private static void SetBit(ref TLink target, int powerOf2Index) => target =
        ↪ Bit.Or(target, Bit.ShiftLeft(_one, powerOf2Index));
62 }
63 }

```

1.85 ./csharp/Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs

```

1  using System.Linq;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.PropertyOperators
9  {
10     public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>, IProperties<TLink, TLink,
        ↪ TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public PropertiesOperator(ILinks<TLink> links) : base(links) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public TLink GetValue(TLink @object, TLink property)
19         {
20             var links = _links;
21             var objectProperty = links.SearchOrDefault(@object, property);
22             if (_equalityComparer.Equals(objectProperty, default))
23             {
24                 return default;
25             }
26             var constants = links.Constants;
27             var valueLink = links.All(constants.Any, objectProperty).SingleOrDefault();
28             if (valueLink == null)
29             {
30                 return default;
31             }
32             return links.GetTarget(valueLink[constants.IndexPart]);
33         }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public void SetValue(TLink @object, TLink property, TLink value)
37         {
38             var links = _links;
39             var objectProperty = links.GetOrCreate(@object, property);
40             links.DeleteMany(links.AllIndices(links.Constants.Any, objectProperty));
41             links.GetOrCreate(objectProperty, value);
42         }
43     }
44 }

```

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

```

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

```

```

9 public class PropertyOperator<TLink> : LinksOperatorBase<TLink>, IProperty<TLink, TLink>
10 {
11     private static readonly EqualityComparer<TLink> _equalityComparer =
12         ↪ EqualityComparer<TLink>.Default;
13
14     private readonly TLink _propertyMarker;
15     private readonly TLink _propertyValueMarker;
16
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
19         ↪ propertyValueMarker) : base(links)
20     {
21         _propertyMarker = propertyMarker;
22         _propertyValueMarker = propertyValueMarker;
23     }
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     public TLink Get(TLink link)
27     {
28         var property = _links.SearchOrDefault(link, _propertyMarker);
29         return GetValue(GetContainer(property));
30     }
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     private TLink GetContainer(TLink property)
34     {
35         var valueContainer = default(TLink);
36         if (_equalityComparer.Equals(property, default))
37         {
38             return valueContainer;
39         }
40         var links = _links;
41         var constants = links.Constants;
42         var countinueConstant = constants.Continue;
43         var breakConstant = constants.Break;
44         var anyConstant = constants.Any;
45         var query = new Link<TLink>(anyConstant, property, anyConstant);
46         links.Each(candidate =>
47         {
48             var candidateTarget = links.GetTarget(candidate);
49             var valueTarget = links.GetTarget(candidateTarget);
50             if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
51             {
52                 valueContainer = links.GetIndex(candidate);
53                 return breakConstant;
54             }
55             return countinueConstant;
56         }, query);
57         return valueContainer;
58     }
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
62         ↪ ? default : _links.GetTarget(container);
63
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     public void Set(TLink link, TLink value)
66     {
67         var links = _links;
68         var property = links.GetOrCreate(link, _propertyMarker);
69         var container = GetContainer(property);
70         if (_equalityComparer.Equals(container, default))
71         {
72             links.GetOrCreate(property, value);
73         }
74         else
75         {
76             links.Update(container, property, value);
77         }
78     }
79 }

```

1.87 ./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
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         private void HalveSequence(ICollection<TLink> destination, ICollection<TLink> source, int length)
46         {
47             var loopedLength = length - (length % 2);
48             for (var i = 0; i < loopedLength; i += 2)
49             {
50                 destination[i / 2] = _links.GetOrCreate(source[i], source[i + 1]);
51             }
52             if (length > loopedLength)
53             {
54                 destination[length / 2] = source[length - 1];
55             }
56         }
57     }

```

1.88 ./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     }

```



```

22 private static readonly EqualityComparer<TLink> _equalityComparer =
    ↳ EqualityComparer<TLink>.Default;
23 private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
24
25 private static readonly TLink _zero = default;
26 private static readonly TLink _one = Arithmetic.Increment(_zero);
27
28 private readonly IConverter<IList<TLink>, TLink> _baseConverter;
29 private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
30 private readonly TLink _minFrequencyToCompress;
31 private readonly bool _doInitialFrequenciesIncrement;
32 private Doublet<TLink> _maxDoublet;
33 private LinkFrequency<TLink> _maxDoubletData;
34
35 private struct HalfDoublet
36 {
37     public TLink Element;
38     public LinkFrequency<TLink> DoubletData;
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
42     {
43         Element = element;
44         DoubletData = doubletData;
45     }
46
47     public override string ToString() => $"{Element}: ({DoubletData})";
48 }
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
52     : this(links, baseConverter, doubletFrequenciesCache, _one, true) { }
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
    ↳ doInitialFrequenciesIncrement)
56     : this(links, baseConverter, doubletFrequenciesCache, _one,
    ↳ doInitialFrequenciesIncrement) { }
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
    ↳ 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(IList<TLink> source) =>
    ↳ _baseConverter.Convert(Compress(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 IList<TLink> Compress(IList<TLink> sequence)
82 {
83     if (sequence.IsNullOrEmpty())
84     {
85         return null;
86     }
87     if (sequence.Count == 1)
88     {
89         return sequence;
90     }
91     if (sequence.Count == 2)
92     {
93         return new[] { _links.GetOrCreate(sequence[0], sequence[1]) };

```

```

94     }
95     // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
96     var copy = new HalfDoublet[sequence.Count];
97     Doublet<TLink> doublet = default;
98     for (var i = 1; i < sequence.Count; i++)
99     {
100         doublet = new Doublet<TLink>(sequence[i - 1], sequence[i]);
101         LinkFrequency<TLink> data;
102         if (_doInitialFrequenciesIncrement)
103         {
104             data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
105         }
106         else
107         {
108             data = _doubletFrequenciesCache.GetFrequency(ref doublet);
109             if (data == null)
110             {
111                 throw new NotSupportedException("If you ask not to increment
112                     ↪ frequencies, it is expected that all frequencies for the sequence
113                     ↪ are prepared.");
114             }
115         }
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 /// <remarks>
135 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
136 /// </remarks>
137 [MethodImpl(MethodImplOptions.AggressiveInlining)]
138 private int ReplaceDoublets(HalfDoublet[] copy)
139 {
140     var oldLength = copy.Length;
141     var newLength = copy.Length;
142     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
143     {
144         var maxDoubletSource = _maxDoublet.Source;
145         var maxDoubletTarget = _maxDoublet.Target;
146         if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
147         {
148             _maxDoubletData.Link = _links.GetOrCreate(maxDoubletSource,
149                 ↪ maxDoubletTarget);
150         }
151         var maxDoubletReplacementLink = _maxDoubletData.Link;
152         oldLength--;
153         var oldLengthMinusTwo = oldLength - 1;
154         // Substitute all usages
155         int w = 0, r = 0; // (r == read, w == write)
156         for (; r < oldLength; r++)
157         {
158             if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
159                 ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
160             {
161                 if (r > 0)
162                 {
163                     var previous = copy[w - 1].Element;
164                     copy[w - 1].DoubletData.DecrementFrequency();
165                     copy[w - 1].DoubletData =
166                         ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
167                             ↪ maxDoubletReplacementLink);
168                 }
169                 if (r < oldLengthMinusTwo)
170                 {
171                     var next = copy[r + 2].Element;

```

```

166         copy[r + 1].DoubletData.DecrementFrequency();
167         copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
        ↪ xDoubletReplacementLink,
        ↪ next);
168     }
169     copy[w++].Element = maxDoubletReplacementLink;
170     r++;
171     newLength--;
172 }
173 else
174 {
175     copy[w++] = copy[r];
176 }
177 }
178 if (w < newLength)
179 {
180     copy[w] = copy[r];
181 }
182 oldLength = newLength;
183 ResetMaxDoublet();
184 UpdateMaxDoublet(copy, newLength);
185 }
186 return newLength;
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 ||
    ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
    ↪ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
    ↪ _maxDoublet.Target)))
213     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
214         (_comparer.Compare(maxFrequency, frequency) < 0 ||
        ↪ (_equalityComparer.Equals(maxFrequency, frequency) &&
        ↪ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
        ↪ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
        ↪ better stability and better compression on sequent data and even on random
        ↪ numbers data (but gives collisions anyway) */
215     {
216         _maxDoublet = doublet;
217         _maxDoubletData = data;
218     }
219 }
220 }
221 }

```

1.89 ./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>,
    ↪ IConverter<IList<TLink>, TLink>
10     {

```

```

11     [MethodImpl(MethodImplOptions.AggressiveInlining)]
12     protected LinksListToSequenceConverterBase(ILinks<TLink> links) : base(links) { }
13
14     [MethodImpl(MethodImplOptions.AggressiveInlining)]
15     public abstract TLink Convert(IList<TLink> source);
16 }
17 }

```

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

```

1  using System.Collections.Generic;
2  using System.Linq;
3  using System.Runtime.CompilerServices;
4  using Platform.Converters;
5  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
6  using Platform.Data.Doublets.Sequences.Frequencies.Counters;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Sequences.Converters
11 {
12     public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↪ EqualityComparer<TLink>.Default;
16         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
17
18         private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
22             ↪ sequenceToItsLocalElementLevelsConverter) : base(links)
23             => _sequenceToItsLocalElementLevelsConverter =
24                 ↪ sequenceToItsLocalElementLevelsConverter;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public OptimalVariantConverter(ILinks<TLink> links, LinkFrequenciesCache<TLink>
28             ↪ linkFrequenciesCache)
29             : this(links, new SequenceToItsLocalElementLevelsConverter<TLink>(links, new Frequen
30                 ↪ ciesCacheBasedLinkToItsFrequencyNumberConverter<TLink>(linkFrequenciesCache))) { }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public OptimalVariantConverter(ILinks<TLink> links)
34             : this(links, new LinkFrequenciesCache<TLink>(links, new
35                 ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links))) { }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public override TLink Convert(IList<TLink> sequence)
39         {
40             var length = sequence.Count;
41             if (length == 1)
42             {
43                 return sequence[0];
44             }
45             if (length == 2)
46             {
47                 return _links.GetOrCreate(sequence[0], sequence[1]);
48             }
49             sequence = sequence.ToArray();
50             var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
51             while (length > 2)
52             {
53                 var levelRepeat = 1;
54                 var currentLevel = levels[0];
55                 var previousLevel = levels[0];
56                 var skipOnce = false;
57                 var w = 0;
58                 for (var i = 1; i < length; i++)
59                 {
60                     if (_equalityComparer.Equals(currentLevel, levels[i]))
61                     {
62                         levelRepeat++;
63                         skipOnce = false;
64                         if (levelRepeat == 2)
65                         {
66                             sequence[w] = _links.GetOrCreate(sequence[i - 1], sequence[i]);
67                             var newLevel = i >= length - 1 ?
68                                 ↪ GetPreviousLowerThanCurrentOrCurrent(previousLevel,
69                                     ↪ currentLevel) :
70                                 ↪ i < 2 ?

```

```

64         GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
65         GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
        ↪ currentLevel, levels[i + 1]);
66         levels[w] = newLevel;
67         previousLevel = currentLevel;
68         w++;
69         levelRepeat = 0;
70         skipOnce = true;
71     }
72     else if (i == length - 1)
73     {
74         sequence[w] = sequence[i];
75         levels[w] = levels[i];
76         w++;
77     }
78 }
79 else
80 {
81     currentLevel = levels[i];
82     levelRepeat = 1;
83     if (skipOnce)
84     {
85         skipOnce = false;
86     }
87     else
88     {
89         sequence[w] = sequence[i - 1];
90         levels[w] = levels[i - 1];
91         previousLevel = levels[w];
92         w++;
93     }
94     if (i == length - 1)
95     {
96         sequence[w] = sequence[i];
97         levels[w] = levels[i];
98         w++;
99     }
100 }
101 }
102 length = w;
103 }
104 return _links.GetOrCreate(sequence[0], sequence[1]);
105 }
106
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 }

```

1.91 ./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 }

```

1.92 ./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 }

```

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

```

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

```

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

```

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

```

1.95 ./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 }

```

```

17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
19     }
20 }

```

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

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Interfaces;
6  using Platform.Collections;
7  using Platform.Collections.Lists;
8  using Platform.Collections.Segments;
9  using Platform.Collections.Segments.Walkers;
10 using Platform.Singletons;
11 using Platform.Converters;
12 using Platform.Data.Doublets.Unicode;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     public class DuplicateSegmentsProvider<TLink> :
19         ↳ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
20         ↳ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
21     {
22         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
23             ↳ UncheckedConverter<TLink, long>.Default;
24         private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
25             ↳ UncheckedConverter<TLink, ulong>.Default;
26         private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
27             ↳ UncheckedConverter<ulong, TLink>.Default;
28
29         private readonly ILinks<TLink> _links;
30         private readonly ILinks<TLink> _sequences;
31         private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
32         private BitString _visited;
33
34         private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
35             ↳ IList<TLink>>>
36         {
37             private readonly IListEqualityComparer<TLink> _listComparer;
38
39             public ItemEquilityComparer() => _listComparer =
40                 ↳ Default<IListEqualityComparer<TLink>>.Instance;
41
42             [MethodImpl(MethodImplOptions.AggressiveInlining)]
43             public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
44                 ↳ KeyValuePair<IList<TLink>, IList<TLink>> right) =>
45                 ↳ _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
46                 ↳ right.Value);
47
48             [MethodImpl(MethodImplOptions.AggressiveInlining)]
49             public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
50                 ↳ (_listComparer.GetHashCode(pair.Key),
51                 ↳ _listComparer.GetHashCode(pair.Value)).GetHashCode();
52         }
53
54         private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
55         {
56             private readonly IListComparer<TLink> _listComparer;
57
58             [MethodImpl(MethodImplOptions.AggressiveInlining)]
59             public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
60
61             [MethodImpl(MethodImplOptions.AggressiveInlining)]
62             public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
63                 ↳ KeyValuePair<IList<TLink>, IList<TLink>> right)
64             {
65                 var intermediateResult = _listComparer.Compare(left.Key, right.Key);
66                 if (intermediateResult == 0)
67                 {
68                     intermediateResult = _listComparer.Compare(left.Value, right.Value);
69                 }
70                 return intermediateResult;
71             }
72         }
73
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

62 public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
63     : base(minimumStringSegmentLength: 2)
64 {
65     _links = links;
66     _sequences = sequences;
67 }
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
71 {
72     _groups = new HashSet<KeyValuePair<IList<TLink>,
73     ↪ IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
74     var links = _links;
75     var count = links.Count();
76     _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
77     links.Each(link =>
78     {
79         var linkIndex = links.GetIndex(link);
80         var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
81         var constants = links.Constants;
82         if (!_visited.Get(linkBitIndex))
83         {
84             var sequenceElements = new List<TLink>();
85             var filler = new ListFiller<TLink, TLink>(sequenceElements, constants.Break);
86             _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
87             ↪ LinkAddress<TLink>(linkIndex));
88             if (sequenceElements.Count > 2)
89             {
90                 WalkAll(sequenceElements);
91             }
92             return constants.Continue;
93         });
94     var resultList = _groups.ToList();
95     var comparer = Default<ItemComparer>.Instance;
96     resultList.Sort(comparer);
97     #if DEBUG
98     foreach (var item in resultList)
99     {
100         PrintDuplicates(item);
101     }
102     #endif
103     return resultList;
104 }
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
108 ↪ length) => new Segment<TLink>(elements, offset, length);
109
110 [MethodImpl(MethodImplOptions.AggressiveInlining)]
111 protected override void OnDuplicateFound(Segment<TLink> segment)
112 {
113     var duplicates = CollectDuplicatesForSegment(segment);
114     if (duplicates.Count > 1)
115     {
116         _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
117         ↪ duplicates));
118     }
119 }
120
121 [MethodImpl(MethodImplOptions.AggressiveInlining)]
122 private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
123 {
124     var duplicates = new List<TLink>();
125     var readAsElement = new HashSet<TLink>();
126     var restrictions = segment.ShiftRight();
127     var constants = _links.Constants;
128     restrictions[0] = constants.Any;
129     _sequences.Each(sequence =>
130     {
131         var sequenceIndex = sequence[constants.IndexPart];
132         duplicates.Add(sequenceIndex);
133         readAsElement.Add(sequenceIndex);
134         return constants.Continue;
135     }, restrictions);
136     if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
137     {
138         return new List<TLink>();
139     }

```

```

137         foreach (var duplicate in duplicates)
138         {
139             var duplicateBitIndex = _addressToInt64Converter.Convert(duplicate);
140             _visited.Set(duplicateBitIndex);
141         }
142         if (_sequences is Sequences sequencesExperiments)
143         {
144             var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H_
145                 ↪ ashSet<ulong>)(object)readAsElement,
146                 ↪ (IList<ulong>)segment);
147             foreach (var partiallyMatchedSequence in partiallyMatched)
148             {
149                 var sequenceIndex =
150                     ↪ _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
151                 duplicates.Add(sequenceIndex);
152             }
153         }
154         duplicates.Sort();
155         return duplicates;
156     }
157 }
158
159 [MethodImpl(MethodImplOptions.AggressiveInlining)]
160 private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
161 {
162     if (!(_links is ILinks<ulong> ulongLinks))
163     {
164         return;
165     }
166     var duplicatesKey = duplicatesItem.Key;
167     var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
168     Console.WriteLine($"{> {keyString} ({string.Join(", ", duplicatesKey)})");
169     var duplicatesList = duplicatesItem.Value;
170     for (int i = 0; i < duplicatesList.Count; i++)
171     {
172         var sequenceIndex = _addressToUInt64Converter.Convert(duplicatesList[i]);
173         var formattedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
174             ↪ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
175             ↪ UnicodeMap.IsCharLink(link.Index) ?
176             ↪ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
177         Console.WriteLine(formattedSequenceStructure);
178         var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
179             ↪ ulongLinks);
180         Console.WriteLine(sequenceString);
181     }
182     Console.WriteLine();
183 }
184 }
185 }
186 }
187 }

```

1.97 ./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)

```

```

28         : base(links)
29     {
30         _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
31             ↳ DoubletComparer<TLink>.Default);
32         _frequencyCounter = frequencyCounter;
33     }
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
37     {
38         var doublet = new Doublet<TLink>(source, target);
39         return GetFrequency(ref doublet);
40     }
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
44     {
45         _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
46         return data;
47     }
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     public void IncrementFrequencies(IList<TLink> sequence)
51     {
52         for (var i = 1; i < sequence.Count; i++)
53         {
54             IncrementFrequency(sequence[i - 1], sequence[i]);
55         }
56     }
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
60     {
61         var doublet = new Doublet<TLink>(source, target);
62         return IncrementFrequency(ref doublet);
63     }
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     public void PrintFrequencies(IList<TLink> sequence)
67     {
68         for (var i = 1; i < sequence.Count; i++)
69         {
70             PrintFrequency(sequence[i - 1], sequence[i]);
71         }
72     }
73
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     public void PrintFrequency(TLink source, TLink target)
76     {
77         var number = GetFrequency(source, target).Frequency;
78         Console.WriteLine("{0},{1}) - {2}", source, target, number);
79     }
80
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
83     {
84         if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
85         {
86             data.IncrementFrequency();
87         }
88         else
89         {
90             var link = _links.SearchOrDefault(doublet.Source, doublet.Target);
91             data = new LinkFrequency<TLink>(_one, link);
92             if (!_equalityComparer.Equals(link, default))
93             {
94                 data.Frequency = Arithmetic.Add(data.Frequency,
95                     ↳ _frequencyCounter.Count(link));
96             }
97             _doubletsCache.Add(doublet, data);
98         }
99         return data;
100     }
101
102     [MethodImpl(MethodImplOptions.AggressiveInlining)]
103     public void ValidateFrequencies()
104     {
105         foreach (var entry in _doubletsCache)

```

```

104     {
105         var value = entry.Value;
106         var linkIndex = value.Link;
107         if (!_equalityComparer.Equals(linkIndex, default))
108         {
109             var frequency = value.Frequency;
110             var count = _frequencyCounter.Count(linkIndex);
111             // TODO: Why `frequency` always greater than `count` by 1?
112             if (((_comparer.Compare(frequency, count) > 0) &&
113                 ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count), _one) > 0))
114                 || ((_comparer.Compare(count, frequency) > 0) &&
115                 ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency), _one) > 0)))
116             {
117                 throw new InvalidOperationException("Frequencies validation failed.");
118             }
119             //else
120             //{
121             //    if (value.Frequency > 0)
122             //    {
123             //        var frequency = value.Frequency;
124             //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
125             //        var count = _countLinkFrequency(linkIndex);
126             //        if ((frequency > count && frequency - count > 1) || (count > frequency
127             ↪ && count - frequency > 1))
128             //            throw new InvalidOperationException("Frequencies validation
129             ↪ failed.");
130             //    }
131             //}
132         }
133     }

```

1.98 ./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 }

```

1.99 ./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>

```

```

9      {
10         private readonly LinkFrequenciesCache<TLink> _cache;
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public
14             ↪ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
15             ↪ cache) => _cache = cache;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
19     }
20 }

```

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

```

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

```

1.101 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4  using Platform.Numbers;
5  using Platform.Data.Sequences;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
10 {
11     public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↪ EqualityComparer<TLink>.Default;
15         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
16
17         protected readonly ILinks<TLink> _links;
18         protected readonly TLink _sequenceLink;
19         protected readonly TLink _symbol;
20         protected TLink _total;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
24             ↪ TLink symbol)
25         {
26             _links = links;
27             _sequenceLink = sequenceLink;
28             _symbol = symbol;
29             _total = default;
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public virtual TLink Count()
34         {
35             if (_comparer.Compare(_total, default) > 0)

```

```

34     {
35         return _total;
36     }
37     StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
38         ↪ IsElement, VisitElement);
39     return _total;
40 }
41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
43     ↪ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
44     ↪ IsPartialPoint
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 private bool VisitElement(TLink element)
47 {
48     if (_equalityComparer.Equals(element, _symbol))
49     {
50         _total = Arithmetic.Increment(_total);
51     }
52     return true;
53 }
54 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Interfaces;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7 {
8     public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
9     {
10         private readonly ILinks<TLink> _links;
11         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
15             ↪ ICriterionMatcher<TLink> markedSequenceMatcher)
16         {
17             _links = links;
18             _markedSequenceMatcher = markedSequenceMatcher;
19         }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public TLink Count(TLink argument) => new
23             ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
24             ↪ _markedSequenceMatcher, argument).Count();
25     }
26 }

```

1.103 ./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         }
27     }
28 }

```

```

22         _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
23     }
24 }
25 }

```

1.104 ./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 }

```

1.105 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4  using Platform.Numbers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
9  {
10     public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↳ EqualityComparer<TLink>.Default;
14         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
15
16         protected readonly ILinks<TLink> _links;
17         protected readonly TLink _symbol;
18         protected readonly HashSet<TLink> _visits;
19         protected TLink _total;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
23         {
24             _links = links;
25             _symbol = symbol;
26             _visits = new HashSet<TLink>();
27             _total = default;
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public TLink Count()
32         {
33             if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
34             {
35                 return _total;
36             }
37             CountCore(_symbol);
38             return _total;
39         }
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         private void CountCore(TLink link)
43         {
44             var any = _links.Constants.Any;
45             if (_equalityComparer.Equals(_links.Count(any, link), default))
46             {
47                 CountSequenceSymbolFrequency(link);
48             }
49             else
50             {
51                 _links.Each(EachElementHandler, any, link);
52             }
53         }
54     }
55 }

```

```

53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected virtual void CountSequenceSymbolFrequency(TLink link)
55     {
56         var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
57             ↪ link, _symbol);
58         _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
59     }
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     private TLink EachElementHandler(IList<TLink> doublet)
63     {
64         var constants = _links.Constants;
65         var doubletIndex = doublet[constants.IndexPart];
66         if (_visits.Add(doubletIndex))
67         {
68             CountCore(doubletIndex);
69         }
70         return constants.Continue;
71     }
72 }
73 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4  using Platform.Converters;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.HeightProviders
9  {
10     public class CachedSequenceHeightProvider<TLink> : ISequenceHeightProvider<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↪ EqualityComparer<TLink>.Default;
14
15         private readonly TLink _heightPropertyMarker;
16         private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
17         private readonly IConverter<TLink> _addressToUnaryNumberConverter;
18         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
19         private readonly IProperties<TLink, TLink, TLink> _propertyOperator;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public CachedSequenceHeightProvider(
23             ISequenceHeightProvider<TLink> baseHeightProvider,
24             IConverter<TLink> addressToUnaryNumberConverter,
25             IConverter<TLink> unaryNumberToAddressConverter,
26             TLink heightPropertyMarker,
27             IProperties<TLink, TLink, TLink> propertyOperator)
28         {
29             _heightPropertyMarker = heightPropertyMarker;
30             _baseHeightProvider = baseHeightProvider;
31             _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
32             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
33             _propertyOperator = propertyOperator;
34
35             [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     }

```


1.107 ./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 }
```

1.108 ./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 }
```

1.109 ./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     }
```

```

32     private bool IsIndexedWithIncrement(TLink source, TLink target)
33     {
34         var frequency = _cache.GetFrequency(source, target);
35         if (frequency == null)
36         {
37             return false;
38         }
39         var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
40         if (indexed)
41         {
42             _cache.IncrementFrequency(source, target);
43         }
44         return indexed;
45     }
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     public bool MightContain(ICollection<TLink> sequence)
49     {
50         var indexed = true;
51         var i = sequence.Count;
52         while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
53         return indexed;
54     }
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     private bool IsIndexed(TLink source, TLink target)
58     {
59         var frequency = _cache.GetFrequency(source, target);
60         if (frequency == null)
61         {
62             return false;
63         }
64         return !_equalityComparer.Equals(frequency.Frequency, default);
65     }
66 }
67 }

```

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

```

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

```

```

39     private bool IsIndexedWithIncrement(TLink source, TLink target)
40     {
41         var link = _links.SearchOrDefault(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 }

```

1.111 ./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 }

```

1.112 ./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;

```

```

32         var i = sequence.Count;
33         while (--i >= 1 && (indexed =
           ↳ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
           ↳ default))) { }
34         return indexed;
35     }
36 }
37 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Indexes
7  {
8      public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↳ EqualityComparer<TLink>.Default;
11
12         private readonly ISynchronizedLinks<TLink> _links;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public bool Add(ICollection<TLink> sequence)
19         {
20             var indexed = true;
21             var i = sequence.Count;
22             var links = _links.Unsync;
23             _links.SyncRoot.ExecuteReadOperation(() =>
24             {
25                 while (--i >= 1 && (indexed =
                   ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                   ↳ sequence[i]), default))) { }
26             });
27             if (!indexed)
28             {
29                 _links.SyncRoot.ExecuteWriteOperation(() =>
30                 {
31                     for (; i >= 1; i--)
32                     {
33                         links.GetOrCreate(sequence[i - 1], sequence[i]);
34                     }
35                 });
36             }
37             return indexed;
38         }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public bool MightContain(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 =
                   ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                   ↳ sequence[i]), default))) { }
49                 return indexed;
50             });
51         }
52     }
53 }

```

1.114 ./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 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using System.Linq;
5  using System.Text;
6  using Platform.Collections;
7  using Platform.Collections.Sets;
8  using Platform.Collections.Stacks;
9  using Platform.Data.Exceptions;
10 using Platform.Data.Sequences;
11 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
12 using Platform.Data.Doublets.Sequences.Walkers;
13 using LinkIndex = System.UInt64;
14 using Stack = System.Collections.Generic.Stack<ulong>;
15
16 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
17
18 namespace Platform.Data.Doublets.Sequences
19 {
20     partial class Sequences
21     {
22         #region Create All Variants (Not Practical)
23
24         /// <remarks>
25         /// Number of links that is needed to generate all variants for
26         /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
27         /// </remarks>
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public ulong[] CreateAllVariants2(ulong[] sequence)
30         {
31             return _sync.ExecuteWriteOperation(() =>
32             {
33                 if (sequence.IsNullOrEmpty())
34                 {
35                     return Array.Empty<ulong>();
36                 }
37                 Links.EnsureLinkExists(sequence);
38                 if (sequence.Length == 1)
39                 {
40                     return sequence;
41                 }
42                 return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
43             });
44         }
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
48         {
49             #if DEBUG
50                 if ((stopAt - startAt) < 0)
51                 {
52                     throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
53                     ↪ меньше или равен stopAt");
54                 }
55             #endif
56             if ((stopAt - startAt) == 0)
57             {
58                 return new[] { sequence[startAt] };
59             }
60             if ((stopAt - startAt) == 1)
61             {
62                 return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt]) };
63             }
64             var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
65             var last = 0;
66             for (var splitter = startAt; splitter < stopAt; splitter++)
67             {
68                 var left = CreateAllVariants2Core(sequence, startAt, splitter);
69                 var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
70                 for (var i = 0; i < left.Length; i++)
71                 {

```

```

71         for (var j = 0; j < right.Length; j++)
72         {
73             var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
74             if (variant == Constants.Null)
75             {
76                 throw new NotImplementedException("Creation cancellation is not
77                 ↪ implemented.");
78             }
79             variants[last++] = variant;
80         }
81     }
82     return variants;
83 }
84
85 [MethodImpl(MethodImplOptions.AggressiveInlining)]
86 public List<ulong> CreateAllVariants1(params ulong[] sequence)
87 {
88     return _sync.ExecuteWriteOperation(() =>
89     {
90         if (sequence.IsNullOrEmpty())
91         {
92             return new List<ulong>();
93         }
94         Links.Unsync.EnsureLinkExists(sequence);
95         if (sequence.Length == 1)
96         {
97             return new List<ulong> { sequence[0] };
98         }
99         var results = new
100             ↪ List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
101         return CreateAllVariants1Core(sequence, results);
102     });
103 }
104
105 [MethodImpl(MethodImplOptions.AggressiveInlining)]
106 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
107 {
108     if (sequence.Length == 2)
109     {
110         var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
111         if (link == Constants.Null)
112         {
113             throw new NotImplementedException("Creation cancellation is not
114             ↪ implemented.");
115         }
116         results.Add(link);
117         return results;
118     }
119     var innerSequenceLength = sequence.Length - 1;
120     var innerSequence = new ulong[innerSequenceLength];
121     for (var li = 0; li < innerSequenceLength; li++)
122     {
123         var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
124         if (link == Constants.Null)
125         {
126             throw new NotImplementedException("Creation cancellation is not
127             ↪ implemented.");
128         }
129         for (var isi = 0; isi < li; isi++)
130         {
131             innerSequence[isi] = sequence[isi];
132         }
133         innerSequence[li] = link;
134         for (var isi = li + 1; isi < innerSequenceLength; isi++)
135         {
136             innerSequence[isi] = sequence[isi + 1];
137         }
138         CreateAllVariants1Core(innerSequence, results);
139     }
140     return results;
141 }
142
143 #endregion
144
145 [MethodImpl(MethodImplOptions.AggressiveInlining)]
146 public HashSet<ulong> Each1(params ulong[] sequence)
147 {

```

```

145     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
146     Each1(link =>
147     {
148         if (!visitedLinks.Contains(link))
149         {
150             visitedLinks.Add(link); // изучить почему случаются повторы
151         }
152         return true;
153     }, sequence);
154     return visitedLinks;
155 }
156
157 [MethodImpl(MethodImplOptions.AggressiveInlining)]
158 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
159 {
160     if (sequence.Length == 2)
161     {
162         Links.Unsync.Each(sequence[0], sequence[1], handler);
163     }
164     else
165     {
166         var innerSequenceLength = sequence.Length - 1;
167         for (var li = 0; li < innerSequenceLength; li++)
168         {
169             var left = sequence[li];
170             var right = sequence[li + 1];
171             if (left == 0 && right == 0)
172             {
173                 continue;
174             }
175             var linkIndex = li;
176             ulong[] innerSequence = null;
177             Links.Unsync.Each(doublet =>
178             {
179                 if (innerSequence == null)
180                 {
181                     innerSequence = new ulong[innerSequenceLength];
182                     for (var isi = 0; isi < linkIndex; isi++)
183                     {
184                         innerSequence[isi] = sequence[isi];
185                     }
186                     for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
187                     {
188                         innerSequence[isi] = sequence[isi + 1];
189                     }
190                 }
191                 innerSequence[linkIndex] = doublet[Constants.IndexPart];
192                 Each1(handler, innerSequence);
193                 return Constants.Continue;
194             }, Constants.Any, left, right);
195         }
196     }
197 }
198
199 [MethodImpl(MethodImplOptions.AggressiveInlining)]
200 public HashSet<ulong> EachPart(params ulong[] sequence)
201 {
202     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
203     EachPartCore(link =>
204     {
205         var linkIndex = link[Constants.IndexPart];
206         if (!visitedLinks.Contains(linkIndex))
207         {
208             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
209         }
210         return Constants.Continue;
211     }, sequence);
212     return visitedLinks;
213 }
214
215 [MethodImpl(MethodImplOptions.AggressiveInlining)]
216 public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
217 {
218     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
219     EachPartCore(link =>
220     {
221         var linkIndex = link[Constants.IndexPart];
222         if (!visitedLinks.Contains(linkIndex))
223         {

```

```

224         visitedLinks.Add(linkIndex); // изучить почему случаются повторы
225         return handler(new LinkAddress<LinkIndex>(linkIndex));
226     }
227     return Constants.Continue;
228 }, sequence);
229 }
230
231 [MethodImpl(MethodImplOptions.AggressiveInlining)]
232 private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
    ↪ sequence)
233 {
234     if (sequence.IsNullOrEmpty())
235     {
236         return;
237     }
238     Links.EnsureLinkIsAnyOrExists(sequence);
239     if (sequence.Length == 1)
240     {
241         var link = sequence[0];
242         if (link > 0)
243         {
244             handler(new LinkAddress<LinkIndex>(link));
245         }
246         else
247         {
248             Links.Each(Constants.Any, Constants.Any, handler);
249         }
250     }
251     else if (sequence.Length == 2)
252     {
253         // _links.Each(sequence[0], sequence[1], handler);
254         //   o_|      x_o ...
255         // x_|      |___|
256         Links.Each(sequence[1], Constants.Any, doublet =>
257         {
258             var match = Links.SearchOrDefault(sequence[0], doublet);
259             if (match != Constants.Null)
260             {
261                 handler(new LinkAddress<LinkIndex>(match));
262             }
263             return true;
264         });
265         // |_x      ... x_o
266         // |_o      |___|
267         Links.Each(Constants.Any, sequence[0], doublet =>
268         {
269             var match = Links.SearchOrDefault(doublet, sequence[1]);
270             if (match != 0)
271             {
272                 handler(new LinkAddress<LinkIndex>(match));
273             }
274             return true;
275         });
276         //           .x o_.
277         //           |___|
278         PartialStepRight(x => handler(x), sequence[0], sequence[1]);
279     }
280     else
281     {
282         throw new NotImplementedException();
283     }
284 }
285
286 [MethodImpl(MethodImplOptions.AggressiveInlining)]
287 private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
288 {
289     Links.Unsync.Each(Constants.Any, left, doublet =>
290     {
291         StepRight(handler, doublet, right);
292         if (left != doublet)
293         {
294             PartialStepRight(handler, doublet, right);
295         }
296         return true;
297     });
298 }
299
300 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

301 private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
302 {
303     Links.Unsync.Each(left, Constants.Any, rightStep =>
304     {
305         TryStepRightUp(handler, right, rightStep);
306         return true;
307     });
308 }
309
310 [MethodImpl(MethodImplOptions.AggressiveInlining)]
311 private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
312 ↪ stepFrom)
313 {
314     var upStep = stepFrom;
315     var firstSource = Links.Unsync.GetTarget(upStep);
316     while (firstSource != right && firstSource != upStep)
317     {
318         upStep = firstSource;
319         firstSource = Links.Unsync.GetSource(upStep);
320     }
321     if (firstSource == right)
322     {
323         handler(new LinkAddress<LinkIndex>(stepFrom));
324     }
325 }
326
327 // TODO: Test
328 [MethodImpl(MethodImplOptions.AggressiveInlining)]
329 private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
330 {
331     Links.Unsync.Each(right, Constants.Any, doublet =>
332     {
333         StepLeft(handler, left, doublet);
334         if (right != doublet)
335         {
336             PartialStepLeft(handler, left, doublet);
337         }
338         return true;
339     });
340 }
341
342 [MethodImpl(MethodImplOptions.AggressiveInlining)]
343 private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
344 {
345     Links.Unsync.Each(Constants.Any, right, leftStep =>
346     {
347         TryStepLeftUp(handler, left, leftStep);
348         return true;
349     });
350 }
351
352 [MethodImpl(MethodImplOptions.AggressiveInlining)]
353 private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
354 {
355     var upStep = stepFrom;
356     var firstTarget = Links.Unsync.GetSource(upStep);
357     while (firstTarget != left && firstTarget != upStep)
358     {
359         upStep = firstTarget;
360         firstTarget = Links.Unsync.GetTarget(upStep);
361     }
362     if (firstTarget == left)
363     {
364         handler(new LinkAddress<LinkIndex>(stepFrom));
365     }
366 }
367
368 [MethodImpl(MethodImplOptions.AggressiveInlining)]
369 private bool StartsWith(ulong sequence, ulong link)
370 {
371     var upStep = sequence;
372     var firstSource = Links.Unsync.GetSource(upStep);
373     while (firstSource != link && firstSource != upStep)
374     {
375         upStep = firstSource;
376         firstSource = Links.Unsync.GetSource(upStep);
377     }
378     return firstSource == link;
}

```

```

379 [MethodImpl(MethodImplOptions.AggressiveInlining)]
380 private bool EndsWith(ulong sequence, ulong link)
381 {
382     var upStep = sequence;
383     var lastTarget = Links.Unsync.GetTarget(upStep);
384     while (lastTarget != link && lastTarget != upStep)
385     {
386         upStep = lastTarget;
387         lastTarget = Links.Unsync.GetTarget(upStep);
388     }
389     return lastTarget == link;
390 }
391
392 [MethodImpl(MethodImplOptions.AggressiveInlining)]
393 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
394 {
395     return _sync.ExecuteReadOperation(() =>
396     {
397         var results = new List<ulong>();
398         if (sequence.Length > 0)
399         {
400             Links.EnsureLinkExists(sequence);
401             var firstElement = sequence[0];
402             if (sequence.Length == 1)
403             {
404                 results.Add(firstElement);
405                 return results;
406             }
407             if (sequence.Length == 2)
408             {
409                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
410                 if (doublet != Constants.Null)
411                 {
412                     results.Add(doublet);
413                 }
414                 return results;
415             }
416             var linksInSequence = new HashSet<ulong>(sequence);
417             void handler(IList<LinkIndex> result)
418             {
419                 var resultIndex = result[Links.Constants.IndexPart];
420                 var filterPosition = 0;
421                 StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
422                     ↪ Links.Unsync.GetTarget,
423                     ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
424                     ↪ x =>
425                     {
426                         if (filterPosition == sequence.Length)
427                         {
428                             filterPosition = -2; // Длиннее чем нужно
429                             return false;
430                         }
431                         if (x != sequence[filterPosition])
432                         {
433                             filterPosition = -1;
434                             return false; // Начинается иначе
435                         }
436                         filterPosition++;
437                         return true;
438                     });
439                 if (filterPosition == sequence.Length)
440                 {
441                     results.Add(resultIndex);
442                 }
443             }
444             if (sequence.Length >= 2)
445             {
446                 StepRight(handler, sequence[0], sequence[1]);
447             }
448             var last = sequence.Length - 2;
449             for (var i = 1; i < last; i++)
450             {
451                 PartialStepRight(handler, sequence[i], sequence[i + 1]);
452             }
453             if (sequence.Length >= 3)
454             {

```

```

455         StepLeft(handler, sequence[sequence.Length - 2],
456             ↪ sequence[sequence.Length - 1]);
457     }
458     }
459     return results;
460 });
461 }
462 [MethodImpl(MethodImplOptions.AggressiveInlining)]
463 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
464 {
465     return _sync.ExecuteReadOperation(() =>
466     {
467         var results = new HashSet<ulong>();
468         if (sequence.Length > 0)
469         {
470             Links.EnsureLinkExists(sequence);
471             var firstElement = sequence[0];
472             if (sequence.Length == 1)
473             {
474                 results.Add(firstElement);
475                 return results;
476             }
477             if (sequence.Length == 2)
478             {
479                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
480                 if (doublet != Constants.Null)
481                 {
482                     results.Add(doublet);
483                 }
484                 return results;
485             }
486             var matcher = new Matcher(this, sequence, results, null);
487             if (sequence.Length >= 2)
488             {
489                 StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
490             }
491             var last = sequence.Length - 2;
492             for (var i = 1; i < last; i++)
493             {
494                 PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
495                     ↪ sequence[i + 1]);
496             }
497             if (sequence.Length >= 3)
498             {
499                 StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
500                     ↪ sequence[sequence.Length - 1]);
501             }
502             }
503         return results;
504     });
505 }
506
507 public const int MaxSequenceFormatSize = 200;
508
509 [MethodImpl(MethodImplOptions.AggressiveInlining)]
510 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
511     ↪ => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
512
513 [MethodImpl(MethodImplOptions.AggressiveInlining)]
514 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
515     ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
516     ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
517     ↪ elementToString, insertComma, knownElements));
518
519 [MethodImpl(MethodImplOptions.AggressiveInlining)]
520 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
521     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
522     ↪ LinkIndex[] knownElements)
523 {
524     var linksInSequence = new HashSet<ulong>(knownElements);
525     //var entered = new HashSet<ulong>();
526     var sb = new StringBuilder();
527     sb.Append('(');
528     if (links.Exists(sequenceLink))
529     {
530         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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
550 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
551 ↪ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
552 ↪ knownElements);
553
554 [MethodImpl(MethodImplOptions.AggressiveInlining)]
555 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
556 ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
557 ↪ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
558 ↪ sequenceLink, elementToString, insertComma, knownElements));
559
560 [MethodImpl(MethodImplOptions.AggressiveInlining)]
561 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
562 ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
563 ↪ LinkIndex[] knownElements)
564 {
565     var linksInSequence = new HashSet<ulong>(knownElements);
566     var entered = new HashSet<ulong>();
567     var sb = new StringBuilder();
568     sb.Append('{');
569     if (links.Exists(sequenceLink))
570     {
571         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
572         ↪ x => linksInSequence.Contains(x) || links.IsFullPoint(x),
573         ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
574     {
575         if (insertComma && sb.Length > 1)
576         {
577             sb.Append(',');
578         }
579         if (entered.Contains(element))
580         {
581             sb.Append('{');
582             elementToString(sb, element);
583             sb.Append('}');
584         }
585         else
586         {
587             elementToString(sb, element);
588         }
589         if (sb.Length < MaxSequenceFormatSize)
590         {
591             return true;
592         }
593         sb.Append(insertComma ? ", ..." : "...");
594         return false;
595     });
596     }
597     sb.Append('}');
598     return sb.ToString();
599 }

```

```

592 [MethodImpl(MethodImplOptions.AggressiveInlining)]
593 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
594 {
595     return _sync.ExecuteReadOperation(() =>
596     {
597         if (sequence.Length > 0)
598         {
599             Links.EnsureLinkExists(sequence);
600             var results = new HashSet<ulong>();
601             for (var i = 0; i < sequence.Length; i++)
602             {
603                 AllUsagesCore(sequence[i], results);
604             }
605             var filteredResults = new List<ulong>();
606             var linksInSequence = new HashSet<ulong>(sequence);
607             foreach (var result in results)
608             {
609                 var filterPosition = -1;
610                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
611                 ↪ Links.Unsync.GetTarget,
612                 ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
613                 ↪ x =>
614                 {
615                     if (filterPosition == (sequence.Length - 1))
616                     {
617                         return false;
618                     }
619                     if (filterPosition >= 0)
620                     {
621                         if (x == sequence[filterPosition + 1])
622                         {
623                             filterPosition++;
624                         }
625                         else
626                         {
627                             return false;
628                         }
629                     }
630                     if (filterPosition < 0)
631                     {
632                         if (x == sequence[0])
633                         {
634                             filterPosition = 0;
635                         }
636                     }
637                     return true;
638                 });
639                 if (filterPosition == (sequence.Length - 1))
640                 {
641                     filteredResults.Add(result);
642                 }
643             }
644             return filteredResults;
645         }
646         return new List<ulong>();
647     });
648 }
649 [MethodImpl(MethodImplOptions.AggressiveInlining)]
650 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
651 {
652     return _sync.ExecuteReadOperation(() =>
653     {
654         if (sequence.Length > 0)
655         {
656             Links.EnsureLinkExists(sequence);
657             var results = new HashSet<ulong>();
658             for (var i = 0; i < sequence.Length; i++)
659             {
660                 AllUsagesCore(sequence[i], results);
661             }
662             var filteredResults = new HashSet<ulong>();
663             var matcher = new Matcher(this, sequence, filteredResults, null);
664             matcher.AddAllPartialMatchedToResults(results);
665             return filteredResults;
666         }
667         return new HashSet<ulong>();

```

```

668     });
669 }
670
671 [MethodImpl(MethodImplOptions.AggressiveInlining)]
672 public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
673 ↪ params ulong[] sequence)
674 {
675     return _sync.ExecuteReadOperation(() =>
676     {
677         if (sequence.Length > 0)
678         {
679             Links.EnsureLinkExists(sequence);
680
681             var results = new HashSet<ulong>();
682             var filteredResults = new HashSet<ulong>();
683             var matcher = new Matcher(this, sequence, filteredResults, handler);
684             for (var i = 0; i < sequence.Length; i++)
685             {
686                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
687                 {
688                     return false;
689                 }
690             }
691             return true;
692         }
693         return true;
694     });
695 }
696
697 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
698 //{
699 //    return Sync.ExecuteReadOperation(() =>
700 //    {
701 //        if (sequence.Length > 0)
702 //        {
703 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
704 //
705 //            var firstResults = new HashSet<ulong>();
706 //            var lastResults = new HashSet<ulong>();
707 //
708 //            var first = sequence.First(x => x != LinksConstants.Any);
709 //            var last = sequence.Last(x => x != LinksConstants.Any);
710 //
711 //            AllUsagesCore(first, firstResults);
712 //            AllUsagesCore(last, lastResults);
713 //
714 //            firstResults.IntersectWith(lastResults);
715 //
716 //            //for (var i = 0; i < sequence.Length; i++)
717 //            //    AllUsagesCore(sequence[i], results);
718 //
719 //            var filteredResults = new HashSet<ulong>();
720 //            var matcher = new Matcher(this, sequence, filteredResults, null);
721 //            matcher.AddAllPartialMatchedToResults(firstResults);
722 //            return filteredResults;
723 //        }
724 //
725 //        return new HashSet<ulong>();
726 //    });
727 //}
728
729 [MethodImpl(MethodImplOptions.AggressiveInlining)]
730 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
731 {
732     return _sync.ExecuteReadOperation((Func<HashSet<ulong>>)(() =>
733     {
734         if (sequence.Length > 0)
735         {
736             ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links,
737 ↪ (IList<ulong>)sequence);
738             var firstResults = new HashSet<ulong>();
739             var lastResults = new HashSet<ulong>();
740             var first = sequence.First(x => x != Constants.Any);
741             var last = sequence.Last(x => x != Constants.Any);
742             AllUsagesCore(first, firstResults);
743             AllUsagesCore(last, lastResults);
744             firstResults.IntersectWith(lastResults);
745             //for (var i = 0; i < sequence.Length; i++)
746             //    AllUsagesCore(sequence[i], results);

```

```

745         var filteredResults = new HashSet<ulong>();
746         var matcher = new Matcher(this, sequence, filteredResults, null);
747         matcher.AddAllPartialMatchedToResults(firstResults);
748         return filteredResults;
749     }
750     return new HashSet<ulong>();
751 }));
752 }
753
754 [MethodImpl(MethodImplOptions.AggressiveInlining)]
755 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
756     ↳ IList<ulong> sequence)
757 {
758     return _sync.ExecuteReadOperation(() =>
759     {
760         if (sequence.Count > 0)
761         {
762             Links.EnsureLinkExists(sequence);
763             var results = new HashSet<LinkIndex>();
764             //var nextResults = new HashSet<ulong>();
765             //for (var i = 0; i < sequence.Length; i++)
766             //{
767                 AllUsagesCore(sequence[i], nextResults);
768                 if (results.IsNullOrEmpty())
769                 {
770                     results = nextResults;
771                     nextResults = new HashSet<ulong>();
772                 }
773                 else
774                 {
775                     results.IntersectWith(nextResults);
776                     nextResults.Clear();
777                 }
778             //}
779             var collector1 = new AllUsagesCollector1(Links.Unsync, results);
780             collector1.Collect(Links.Unsync.GetLink(sequence[0]));
781             var next = new HashSet<ulong>();
782             for (var i = 1; i < sequence.Count; i++)
783             {
784                 var collector = new AllUsagesCollector1(Links.Unsync, next);
785                 collector.Collect(Links.Unsync.GetLink(sequence[i]));
786
787                 results.IntersectWith(next);
788                 next.Clear();
789             }
790             var filteredResults = new HashSet<ulong>();
791             var matcher = new Matcher(this, sequence, filteredResults, null,
792                 ↳ readAsElements);
793             matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
794                 ↳ x)); // OrderBy is a Hack
795             return filteredResults;
796         }
797         return new HashSet<ulong>();
798     });
799 }
800
801 // Does not work
802 //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
803 //    ↳ params ulong[] sequence)
804 //{
805 //    var visited = new HashSet<ulong>();
806 //    var results = new HashSet<ulong>();
807 //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
808 //        ↳ true; }, readAsElements);
809 //    var last = sequence.Length - 1;
810 //    for (var i = 0; i < last; i++)
811 //    {
812 //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
813 //    }
814 //    return results;
815 //}
816
817 [MethodImpl(MethodImplOptions.AggressiveInlining)]
818 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
819 {
820     return _sync.ExecuteReadOperation(() =>
821     {
822         if (sequence.Length > 0)

```

```

818 {
819     Links.EnsureLinkExists(sequence);
820     //var firstElement = sequence[0];
821     //if (sequence.Length == 1)
822     //{
823         //    //results.Add(firstElement);
824         //    return results;
825     //}
826     //if (sequence.Length == 2)
827     //{
828         //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
829         //    //if (doublet != Doublets.Links.Null)
830         //    //    results.Add(doublet);
831         //    return results;
832     //}
833     //var lastElement = sequence[sequence.Length - 1];
834     //Func<ulong, bool> handler = x =>
835     //{
836         //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
837         //        results.Add(x);
838         //    return true;
839     //};
840     //if (sequence.Length >= 2)
841         StepRight(handler, sequence[0], sequence[1]);
842     //var last = sequence.Length - 2;
843     //for (var i = 1; i < last; i++)
844         PartialStepRight(handler, sequence[i], sequence[i + 1]);
845     //if (sequence.Length >= 3)
846         StepLeft(handler, sequence[sequence.Length - 2],
847             sequence[sequence.Length - 1]);
848     //if (sequence.Length == 1)
849     //{
850         //    throw new NotImplementedException(); // all sequences, containing
851         //    this element?
852     //}
853     //if (sequence.Length == 2)
854     //{
855         //    var results = new List<ulong>();
856         //    PartialStepRight(results.Add, sequence[0], sequence[1]);
857         //    return results;
858     //}
859     //var matches = new List<List<ulong>>();
860     //var last = sequence.Length - 1;
861     //for (var i = 0; i < last; i++)
862     //{
863         //    var results = new List<ulong>();
864         //    //StepRight(results.Add, sequence[i], sequence[i + 1]);
865         //    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
866         //    if (results.Count > 0)
867             matches.Add(results);
868         //    else
869             return results;
870         //    if (matches.Count == 2)
871         //    {
872             var merged = new List<ulong>();
873             for (var j = 0; j < matches[0].Count; j++)
874                 for (var k = 0; k < matches[1].Count; k++)
875                     CloseInnerConnections(merged.Add, matches[0][j],
876                         matches[1][k]);
877             if (merged.Count > 0)
878                 matches = new List<List<ulong>> { merged };
879             else
880                 return new List<ulong>();
881         //    }
882     //}
883     //if (matches.Count > 0)
884     //{
885         //    var usages = new HashSet<ulong>();
886         //    for (int i = 0; i < sequence.Length; i++)
887         //    {
888             AllUsagesCore(sequence[i], usages);
889         //    }
890         //    //for (int i = 0; i < matches[0].Count; i++)
891         //    //    AllUsagesCore(matches[0][i], usages);
892         //    //usages.UnionWith(matches[0]);
893         //    return usages.ToList();
894     //}

```



```

891         var firstLinkUsages = new HashSet<ulong>();
892         AllUsagesCore(sequence[0], firstLinkUsages);
893         firstLinkUsages.Add(sequence[0]);
894         //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
895         ↪ sequence[0] }; // or all sequences, containing this element?
896         //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
897         ↪ 1).ToList();
898         var results = new HashSet<ulong>();
899         foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
900         ↪ firstLinkUsages, 1))
901         {
902             AllUsagesCore(match, results);
903         }
904         return results.ToList();
905     }
906     return new List<ulong>();
907 });
908
909 /// <remarks>
910 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
911 /// </remarks>
912 [MethodImpl(MethodImplOptions.AggressiveInlining)]
913 public HashSet<ulong> AllUsages(ulong link)
914 {
915     return _sync.ExecuteReadOperation(() =>
916     {
917         var usages = new HashSet<ulong>();
918         AllUsagesCore(link, usages);
919         return usages;
920     });
921 }
922
923 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
924 ↪ той связи с которой начинался поиск (STTTSSSTT),
925 // причём достаточно одного бита для хранения перехода влево или вправо
926 [MethodImpl(MethodImplOptions.AggressiveInlining)]
927 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
928 {
929     bool handler(ulong doublet)
930     {
931         if (usages.Add(doublet))
932         {
933             AllUsagesCore(doublet, usages);
934         }
935         return true;
936     }
937     Links.Unsync.Each(link, Constants.Any, handler);
938     Links.Unsync.Each(Constants.Any, link, handler);
939 }
940
941 [MethodImpl(MethodImplOptions.AggressiveInlining)]
942 public HashSet<ulong> AllBottomUsages(ulong link)
943 {
944     return _sync.ExecuteReadOperation(() =>
945     {
946         var visits = new HashSet<ulong>();
947         var usages = new HashSet<ulong>();
948         AllBottomUsagesCore(link, visits, usages);
949         return usages;
950     });
951 }
952
953 [MethodImpl(MethodImplOptions.AggressiveInlining)]
954 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
955 ↪ usages)
956 {
957     bool handler(ulong doublet)
958     {
959         if (visits.Add(doublet))
960         {
961             AllBottomUsagesCore(doublet, visits, usages);
962         }
963         return true;
964     }
965     if (Links.Unsync.Count(Constants.Any, link) == 0)
966     {
967         usages.Add(link);
968     }
969 }

```

```

964     }
965     else
966     {
967         Links.Unsync.Each(link, Constants.Any, handler);
968         Links.Unsync.Each(Constants.Any, link, handler);
969     }
970 }
971
972 [MethodImpl(MethodImplOptions.AggressiveInlining)]
973 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
974 {
975     if (Options.UseSequenceMarker)
976     {
977         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
978             ↪ Options.MarkedSequenceMatcher, symbol);
979         return counter.Count();
980     }
981     else
982     {
983         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
984             ↪ symbol);
985         return counter.Count();
986     }
987 }
988
989 [MethodImpl(MethodImplOptions.AggressiveInlining)]
990 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
991     ↪ LinkIndex> outerHandler)
992 {
993     bool handler(ulong doublet)
994     {
995         if (usages.Add(doublet))
996         {
997             if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
998             {
999                 return false;
1000             }
1001             if (!AllUsagesCore1(doublet, usages, outerHandler))
1002             {
1003                 return false;
1004             }
1005         }
1006         return true;
1007     }
1008     return Links.Unsync.Each(link, Constants.Any, handler)
1009         && Links.Unsync.Each(Constants.Any, link, handler);
1010 }
1011
1012 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1013 public void CalculateAllUsages(ulong[] totals)
1014 {
1015     var calculator = new AllUsagesCalculator(Links, totals);
1016     calculator.Calculate();
1017 }
1018
1019 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1020 public void CalculateAllUsages2(ulong[] totals)
1021 {
1022     var calculator = new AllUsagesCalculator2(Links, totals);
1023     calculator.Calculate();
1024 }
1025
1026 private class AllUsagesCalculator
1027 {
1028     private readonly SynchronizedLinks<ulong> _links;
1029     private readonly ulong[] _totals;
1030
1031     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1032     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
1033     {
1034         _links = links;
1035         _totals = totals;
1036     }
1037
1038     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1039     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1040         ↪ CalculateCore);
1041
1042     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

1039 private bool CalculateCore(ulong link)
1040 {
1041     if (_totals[link] == 0)
1042     {
1043         var total = 1UL;
1044         _totals[link] = total;
1045         var visitedChildren = new HashSet<ulong>();
1046         bool linkCalculator(ulong child)
1047         {
1048             if (link != child && visitedChildren.Add(child))
1049             {
1050                 total += _totals[child] == 0 ? 1 : _totals[child];
1051             }
1052             return true;
1053         }
1054         _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1055         _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1056         _totals[link] = total;
1057     }
1058     return true;
1059 }
1060
1061 private class AllUsagesCalculator2
1062 {
1063     private readonly SynchronizedLinks<ulong> _links;
1064     private readonly ulong[] _totals;
1065
1066     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1067     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1068     {
1069         _links = links;
1070         _totals = totals;
1071     }
1072
1073     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1074     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1075         ↪ CalculateCore);
1076
1077     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1078     private bool IsElement(ulong link)
1079     {
1080         // _linksInSequence.Contains(link) ||
1081         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1082             ↪ link;
1083     }
1084
1085     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1086     private bool CalculateCore(ulong link)
1087     {
1088         // TODO: Проработать защиту от заикливания
1089         // Основано на SequenceWalker.WalkLeft
1090         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1091         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1092         Func<ulong, bool> isElement = IsElement;
1093         void visitLeaf(ulong parent)
1094         {
1095             if (link != parent)
1096             {
1097                 _totals[parent]++;
1098             }
1099         }
1100         void visitNode(ulong parent)
1101         {
1102             if (link != parent)
1103             {
1104                 _totals[parent]++;
1105             }
1106         }
1107         var stack = new Stack();
1108         var element = link;
1109         if (isElement(element))
1110         {
1111             visitLeaf(element);
1112         }
1113         else
1114         {
1115             while (true)
1116             {

```

```

1116         if (isElement(element))
1117         {
1118             if (stack.Count == 0)
1119             {
1120                 break;
1121             }
1122             element = stack.Pop();
1123             var source = getSource(element);
1124             var target = getTarget(element);
1125             // Обработка элемента
1126             if (isElement(target))
1127             {
1128                 visitLeaf(target);
1129             }
1130             if (isElement(source))
1131             {
1132                 visitLeaf(source);
1133             }
1134             element = source;
1135         }
1136         else
1137         {
1138             stack.Push(element);
1139             visitNode(element);
1140             element = getTarget(element);
1141         }
1142     }
1143 }
1144 _totals[link]++;
1145 return true;
1146 }
1147 }
1148
1149 private class AllUsagesCollector
1150 {
1151     private readonly ILinks<ulong> _links;
1152     private readonly HashSet<ulong> _usages;
1153
1154     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1155     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1156     {
1157         _links = links;
1158         _usages = usages;
1159     }
1160
1161     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1162     public bool Collect(ulong link)
1163     {
1164         if (_usages.Add(link))
1165         {
1166             _links.Each(link, _links.Constants.Any, Collect);
1167             _links.Each(_links.Constants.Any, link, Collect);
1168         }
1169         return true;
1170     }
1171 }
1172
1173 private class AllUsagesCollector1
1174 {
1175     private readonly ILinks<ulong> _links;
1176     private readonly HashSet<ulong> _usages;
1177     private readonly ulong _continue;
1178
1179     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1180     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1181     {
1182         _links = links;
1183         _usages = usages;
1184         _continue = _links.Constants.Continue;
1185     }
1186
1187     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1188     public ulong Collect(ICollection<ulong> link)
1189     {
1190         var linkIndex = _links.GetIndex(link);
1191         if (_usages.Add(linkIndex))
1192         {
1193             _links.Each(Collect, _links.Constants.Any, linkIndex);
1194         }
1195         return _continue;

```

```

1196     }
1197 }
1198
1199 private class AllUsagesCollector2
1200 {
1201     private readonly ILinks<ulong> _links;
1202     private readonly BitString _usages;
1203
1204     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1205     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1206     {
1207         _links = links;
1208         _usages = usages;
1209     }
1210
1211     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1212     public bool Collect(ulong link)
1213     {
1214         if (_usages.Add((long)link))
1215         {
1216             _links.Each(link, _links.Constants.Any, Collect);
1217             _links.Each(_links.Constants.Any, link, Collect);
1218         }
1219         return true;
1220     }
1221 }
1222
1223 private class AllUsagesIntersectingCollector
1224 {
1225     private readonly SynchronizedLinks<ulong> _links;
1226     private readonly HashSet<ulong> _intersectWith;
1227     private readonly HashSet<ulong> _usages;
1228     private readonly HashSet<ulong> _enter;
1229
1230     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1231     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
↪ intersectWith, HashSet<ulong> usages)
1232     {
1233         _links = links;
1234         _intersectWith = intersectWith;
1235         _usages = usages;
1236         _enter = new HashSet<ulong>(); // защита от зацикливания
1237     }
1238
1239     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1240     public bool Collect(ulong link)
1241     {
1242         if (_enter.Add(link))
1243         {
1244             if (_intersectWith.Contains(link))
1245             {
1246                 _usages.Add(link);
1247             }
1248             _links.Unsync.Each(link, _links.Constants.Any, Collect);
1249             _links.Unsync.Each(_links.Constants.Any, link, Collect);
1250         }
1251         return true;
1252     }
1253 }
1254
1255 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1256 private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
↪ right)
1257 {
1258     TryStepLeftUp(handler, left, right);
1259     TryStepRightUp(handler, right, left);
1260 }
1261
1262 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1263 private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
↪ right)
1264 {
1265     // Direct
1266     if (left == right)
1267     {
1268         handler(new LinkAddress<LinkIndex>(left));
1269     }
1270     var doublet = Links.Unsync.SearchOrDefault(left, right);
1271     if (doublet != Constants.Null)

```

```

1272     {
1273         handler(new LinkAddress<LinkIndex>(doublet));
1274     }
1275     // Inner
1276     CloseInnerConnections(handler, left, right);
1277     // Outer
1278     StepLeft(handler, left, right);
1279     StepRight(handler, left, right);
1280     PartialStepRight(handler, left, right);
1281     PartialStepLeft(handler, left, right);
1282 }
1283
1284 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1285 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
1286     ↪ HashSet<ulong> previousMatchings, long startAt)
1287 {
1288     if (startAt >= sequence.Length) // ?
1289     {
1290         return previousMatchings;
1291     }
1292     var secondLinkUsages = new HashSet<ulong>();
1293     AllUsagesCore(sequence[startAt], secondLinkUsages);
1294     secondLinkUsages.Add(sequence[startAt]);
1295     var matchings = new HashSet<ulong>();
1296     var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
1297     //for (var i = 0; i < previousMatchings.Count; i++)
1298     foreach (var secondLinkUsage in secondLinkUsages)
1299     {
1300         foreach (var previousMatching in previousMatchings)
1301         {
1302             //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
1303             ↪ secondLinkUsage);
1304             StepRight(filler.AddFirstAndReturnConstant, previousMatching,
1305             ↪ secondLinkUsage);
1306             TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
1307             ↪ previousMatching);
1308             //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1309             ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1310             ↪ желаемым результатам.
1311             PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
1312             ↪ secondLinkUsage);
1313         }
1314     }
1315     if (matchings.Count == 0)
1316     {
1317         return matchings;
1318     }
1319     return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1320 }
1321
1322 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1323 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
1324     ↪ links, params ulong[] sequence)
1325 {
1326     if (sequence == null)
1327     {
1328         return;
1329     }
1330     for (var i = 0; i < sequence.Length; i++)
1331     {
1332         if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
1333             ↪ !links.Exists(sequence[i]))
1334         {
1335             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1336             ↪ $"patternSequence[{i}]");
1337         }
1338     }
1339 }
1340
1341 // Pattern Matching -> Key To Triggers
1342 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1343 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1344 {
1345     return _sync.ExecuteReadOperation(() =>
1346     {
1347         patternSequence = Simplify(patternSequence);
1348         if (patternSequence.Length > 0)

```

```

1339     {
1340         EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1341         var uniqueSequenceElements = new HashSet<ulong>();
1342         for (var i = 0; i < patternSequence.Length; i++)
1343         {
1344             if (patternSequence[i] != Constants.Any && patternSequence[i] !=
1345                 ↪ ZeroOrMany)
1346             {
1347                 uniqueSequenceElements.Add(patternSequence[i]);
1348             }
1349             var results = new HashSet<ulong>();
1350             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1351             {
1352                 AllUsagesCore(uniqueSequenceElement, results);
1353             }
1354             var filteredResults = new HashSet<ulong>();
1355             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1356             matcher.AddAllPatternMatchedToResults(results);
1357             return filteredResults;
1358         }
1359         return new HashSet<ulong>();
1360     });
1361 }
1362
1363 // Найти все возможные связи между указанным списком связей.
1364 // Находит связи между всеми указанными связями в любом порядке.
1365 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
1366 ↪ несколько раз в последовательности)
1367 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1368 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1369 {
1370     return _sync.ExecuteReadOperation(() =>
1371     {
1372         var results = new HashSet<ulong>();
1373         if (linksToConnect.Length > 0)
1374         {
1375             Links.EnsureLinkExists(linksToConnect);
1376             AllUsagesCore(linksToConnect[0], results);
1377             for (var i = 1; i < linksToConnect.Length; i++)
1378             {
1379                 var next = new HashSet<ulong>();
1380                 AllUsagesCore(linksToConnect[i], next);
1381                 results.IntersectWith(next);
1382             }
1383             return results;
1384         }
1385     });
1386 }
1387 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1388 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1389 {
1390     return _sync.ExecuteReadOperation(() =>
1391     {
1392         var results = new HashSet<ulong>();
1393         if (linksToConnect.Length > 0)
1394         {
1395             Links.EnsureLinkExists(linksToConnect);
1396             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1397             collector1.Collect(linksToConnect[0]);
1398             var next = new HashSet<ulong>();
1399             for (var i = 1; i < linksToConnect.Length; i++)
1400             {
1401                 var collector = new AllUsagesCollector(Links.Unsync, next);
1402                 collector.Collect(linksToConnect[i]);
1403                 results.IntersectWith(next);
1404                 next.Clear();
1405             }
1406             return results;
1407         }
1408     });
1409 }
1410 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1411 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1412 {
1413     return _sync.ExecuteReadOperation(() =>

```

```

1415 {
1416     var results = new HashSet<ulong>();
1417     if (linksToConnect.Length > 0)
1418     {
1419         Links.EnsureLinkExists(linksToConnect);
1420         var collector1 = new AllUsagesCollector(Links, results);
1421         collector1.Collect(linksToConnect[0]);
1422         //AllUsagesCore(linksToConnect[0], results);
1423         for (var i = 1; i < linksToConnect.Length; i++)
1424         {
1425             var next = new HashSet<ulong>();
1426             var collector = new AllUsagesIntersectingCollector(Links, results, next);
1427             collector.Collect(linksToConnect[i]);
1428             //AllUsagesCore(linksToConnect[i], next);
1429             //results.IntersectWith(next);
1430             results = next;
1431         }
1432     }
1433     return results;
1434 });
1435 }
1436
1437 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1438 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1439 {
1440     return _sync.ExecuteReadOperation(() =>
1441     {
1442         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1443         ↪ BitArray((int)_links.Total + 1);
1444         if (linksToConnect.Length > 0)
1445         {
1446             Links.EnsureLinkExists(linksToConnect);
1447             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1448             collector1.Collect(linksToConnect[0]);
1449             for (var i = 1; i < linksToConnect.Length; i++)
1450             {
1451                 var next = new BitString((long)Links.Unsync.Count() + 1); //new
1452                 ↪ BitArray((int)_links.Total + 1);
1453                 var collector = new AllUsagesCollector2(Links.Unsync, next);
1454                 collector.Collect(linksToConnect[i]);
1455                 results = results.And(next);
1456             }
1457         }
1458         return results.GetSetUInt64Indices();
1459     });
1460 }
1461
1462 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1463 private static ulong[] Simplify(ulong[] sequence)
1464 {
1465     // Считаем новый размер последовательности
1466     long newLength = 0;
1467     var zeroOrManyStepped = false;
1468     for (var i = 0; i < sequence.Length; i++)
1469     {
1470         if (sequence[i] == ZeroOrMany)
1471         {
1472             if (zeroOrManyStepped)
1473             {
1474                 continue;
1475             }
1476             zeroOrManyStepped = true;
1477         }
1478         else
1479         {
1480             //if (zeroOrManyStepped) Is it efficient?
1481             zeroOrManyStepped = false;
1482         }
1483         newLength++;
1484     }
1485     // Строим новую последовательность
1486     zeroOrManyStepped = false;
1487     var newSequence = new ulong[newLength];
1488     long j = 0;
1489     for (var i = 0; i < sequence.Length; i++)
1490     {
1491         //var current = zeroOrManyStepped;
1492         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;

```



```

1491         //if (current && zeroOrManyStepped)
1492         //    continue;
1493         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1494         //if (zeroOrManyStepped && newZeroOrManyStepped)
1495         //    continue;
1496         //zeroOrManyStepped = newZeroOrManyStepped;
1497         if (sequence[i] == ZeroOrMany)
1498         {
1499             if (zeroOrManyStepped)
1500             {
1501                 continue;
1502             }
1503             zeroOrManyStepped = true;
1504         }
1505         else
1506         {
1507             //if (zeroOrManyStepped) Is it efficient?
1508             zeroOrManyStepped = false;
1509         }
1510         newSequence[j++] = sequence[i];
1511     }
1512     return newSequence;
1513 }
1514
1515 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1516 public static void TestSimplify()
1517 {
1518     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
1519         ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1520     var simplifiedSequence = Simplify(sequence);
1521 }
1522
1523 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1524 public List<ulong> GetSimilarSequences() => new List<ulong>();
1525
1526 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1527 public void Prediction()
1528 {
1529     //_links
1530     //_sequences
1531 }
1532
1533 #region From Triplets
1534
1535 //public static void DeleteSequence(Link sequence)
1536 //{
1537 //}
1538
1539 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1540 public List<ulong> CollectMatchingSequences(ulong[] links)
1541 {
1542     if (links.Length == 1)
1543     {
1544         throw new InvalidOperationException("Подпоследовательности с одним элементом не
1545             ↪ поддерживаются.");
1546     }
1547     var leftBound = 0;
1548     var rightBound = links.Length - 1;
1549     var left = links[leftBound+1];
1550     var right = links[rightBound-1];
1551     var results = new List<ulong>();
1552     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1553     return results;
1554 }
1555
1556 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1557 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1558     ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1559 {
1560     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1561     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1562     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1563     {
1564         var nextLeftLink = middleLinks[leftBound];
1565         var elements = GetRightElements(leftLink, nextLeftLink);
1566         if (leftBound <= rightBound)
1567         {
1568             for (var i = elements.Length - 1; i >= 0; i--)
1569             {

```

```

1567         var element = elements[i];
1568         if (element != 0)
1569         {
1570             CollectMatchingSequences(element, leftBound + 1, middleLinks,
1571                                     ↪ rightLink, rightBound, ref results);
1572         }
1573     }
1574     else
1575     {
1576         for (var i = elements.Length - 1; i >= 0; i--)
1577         {
1578             var element = elements[i];
1579             if (element != 0)
1580             {
1581                 results.Add(element);
1582             }
1583         }
1584     }
1585 }
1586 else
1587 {
1588     var nextRightLink = middleLinks[rightBound];
1589     var elements = GetLeftElements(rightLink, nextRightLink);
1590     if (leftBound <= rightBound)
1591     {
1592         for (var i = elements.Length - 1; i >= 0; i--)
1593         {
1594             var element = elements[i];
1595             if (element != 0)
1596             {
1597                 CollectMatchingSequences(leftLink, leftBound, middleLinks,
1598                                         ↪ elements[i], rightBound - 1, ref results);
1599             }
1600         }
1601     }
1602     else
1603     {
1604         for (var i = elements.Length - 1; i >= 0; i--)
1605         {
1606             var element = elements[i];
1607             if (element != 0)
1608             {
1609                 results.Add(element);
1610             }
1611         }
1612     }
1613 }
1614
1615 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1616 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1617 {
1618     var result = new ulong[5];
1619     TryStepRight(startLink, rightLink, result, 0);
1620     Links.Each(Constants.Any, startLink, couple =>
1621     {
1622         if (couple != startLink)
1623         {
1624             if (TryStepRight(couple, rightLink, result, 2))
1625             {
1626                 return false;
1627             }
1628         }
1629         return true;
1630     });
1631     if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1632     {
1633         result[4] = startLink;
1634     }
1635     return result;
1636 }
1637
1638 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1639 public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1640 {
1641     var added = 0;
1642     Links.Each(startLink, Constants.Any, couple =>

```

```

1643 {
1644     if (couple != startLink)
1645     {
1646         var coupleTarget = Links.GetTarget(couple);
1647         if (coupleTarget == rightLink)
1648         {
1649             result[offset] = couple;
1650             if (++added == 2)
1651             {
1652                 return false;
1653             }
1654         }
1655         else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
1656             ↪ == Net.And &&
1657         {
1658             result[offset + 1] = couple;
1659             if (++added == 2)
1660             {
1661                 return false;
1662             }
1663         }
1664     }
1665     return true;
1666 });
1667 return added > 0;
1668 }
1669 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1670 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1671 {
1672     var result = new ulong[5];
1673     TryStepLeft(startLink, leftLink, result, 0);
1674     Links.Each(startLink, Constants.Any, couple =>
1675     {
1676         if (couple != startLink)
1677         {
1678             if (TryStepLeft(couple, leftLink, result, 2))
1679             {
1680                 return false;
1681             }
1682         }
1683     });
1684     return true;
1685     if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1686     {
1687         result[4] = leftLink;
1688     }
1689     return result;
1690 }
1691 }
1692 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1693 public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1694 {
1695     var added = 0;
1696     Links.Each(Constants.Any, startLink, couple =>
1697     {
1698         if (couple != startLink)
1699         {
1700             var coupleSource = Links.GetSource(couple);
1701             if (coupleSource == leftLink)
1702             {
1703                 result[offset] = couple;
1704                 if (++added == 2)
1705                 {
1706                     return false;
1707                 }
1708             }
1709             else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1710                 ↪ == Net.And &&
1711             {
1712                 result[offset + 1] = couple;
1713                 if (++added == 2)
1714                 {
1715                     return false;
1716                 }
1717             }
1718         }
1719     });
1720     return true;
1721 });

```

```

1720     return added > 0;
1721 }
1722
1723 #endregion
1724
1725 #region Walkers
1726
1727 public class PatternMatcher : RightSequenceWalker<ulong>
1728 {
1729     private readonly Sequences _sequences;
1730     private readonly ulong[] _patternSequence;
1731     private readonly HashSet<LinkIndex> _linksInSequence;
1732     private readonly HashSet<LinkIndex> _results;
1733
1734     #region Pattern Match
1735
1736     enum PatternBlockType
1737     {
1738         Undefined,
1739         Gap,
1740         Elements
1741     }
1742
1743     struct PatternBlock
1744     {
1745         public PatternBlockType Type;
1746         public long Start;
1747         public long Stop;
1748     }
1749
1750     private readonly List<PatternBlock> _pattern;
1751     private int _patternPosition;
1752     private long _sequencePosition;
1753
1754     #endregion
1755
1756     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1757     public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1758         ↳ HashSet<LinkIndex> results)
1759         : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1760     {
1761         _sequences = sequences;
1762         _patternSequence = patternSequence;
1763         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1764             ↳ _sequences.Constants.Any && x != ZeroOrMany));
1765         _results = results;
1766         _pattern = CreateDetailedPattern();
1767     }
1768
1769     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1770     protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
1771         ↳ base.IsElement(link);
1772
1773     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1774     public bool PatternMatch(LinkIndex sequenceToMatch)
1775     {
1776         _patternPosition = 0;
1777         _sequencePosition = 0;
1778         foreach (var part in Walk(sequenceToMatch))
1779         {
1780             if (!PatternMatchCore(part))
1781             {
1782                 break;
1783             }
1784         }
1785         return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1786             ↳ - 1 && _pattern[_patternPosition].Start == 0);
1787     }
1788
1789     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1790     private List<PatternBlock> CreateDetailedPattern()
1791     {
1792         var pattern = new List<PatternBlock>();
1793         var patternBlock = new PatternBlock();
1794         for (var i = 0; i < _patternSequence.Length; i++)
1795         {
1796             if (patternBlock.Type == PatternBlockType.Undefined)
1797             {
1798                 if (_patternSequence[i] == _sequences.Constants.Any)
1799                 {

```

```

1796         patternBlock.Type = PatternBlockType.Gap;
1797         patternBlock.Start = 1;
1798         patternBlock.Stop = 1;
1799     }
1800     else if (_patternSequence[i] == ZeroOrMany)
1801     {
1802         patternBlock.Type = PatternBlockType.Gap;
1803         patternBlock.Start = 0;
1804         patternBlock.Stop = long.MaxValue;
1805     }
1806     else
1807     {
1808         patternBlock.Type = PatternBlockType.Elements;
1809         patternBlock.Start = i;
1810         patternBlock.Stop = i;
1811     }
1812 }
1813 else if (patternBlock.Type == PatternBlockType.Elements)
1814 {
1815     if (_patternSequence[i] == _sequences.Constants.Any)
1816     {
1817         pattern.Add(patternBlock);
1818         patternBlock = new PatternBlock
1819         {
1820             Type = PatternBlockType.Gap,
1821             Start = 1,
1822             Stop = 1
1823         };
1824     }
1825     else if (_patternSequence[i] == ZeroOrMany)
1826     {
1827         pattern.Add(patternBlock);
1828         patternBlock = new PatternBlock
1829         {
1830             Type = PatternBlockType.Gap,
1831             Start = 0,
1832             Stop = long.MaxValue
1833         };
1834     }
1835     else
1836     {
1837         patternBlock.Stop = i;
1838     }
1839 }
1840 else // patternBlock.Type == PatternBlockType.Gap
1841 {
1842     if (_patternSequence[i] == _sequences.Constants.Any)
1843     {
1844         patternBlock.Start++;
1845         if (patternBlock.Stop < patternBlock.Start)
1846         {
1847             patternBlock.Stop = patternBlock.Start;
1848         }
1849     }
1850     else if (_patternSequence[i] == ZeroOrMany)
1851     {
1852         patternBlock.Stop = long.MaxValue;
1853     }
1854     else
1855     {
1856         pattern.Add(patternBlock);
1857         patternBlock = new PatternBlock
1858         {
1859             Type = PatternBlockType.Elements,
1860             Start = i,
1861             Stop = i
1862         };
1863     }
1864 }
1865 }
1866 if (patternBlock.Type != PatternBlockType.Undefined)
1867 {
1868     pattern.Add(patternBlock);
1869 }
1870 return pattern;
1871 }
1872
1873 // match: search for regexp anywhere in text
1874 //int match(char* regexp, char* text)
1875 //{

```

```

1876 // do
1877 // {
1878 // } while (*text++ != '\0');
1879 // return 0;
1880 //}
1881
1882 // matchhere: search for regexp at beginning of text
1883 //int matchhere(char* regexp, char* text)
1884 //{
1885 //    if (regexp[0] == '\0')
1886 //        return 1;
1887 //    if (regexp[1] == '*')
1888 //        return matchstar(regexp[0], regexp + 2, text);
1889 //    if (regexp[0] == '$' && regexp[1] == '\0')
1890 //        return *text == '\0';
1891 //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1892 //        return matchhere(regexp + 1, text + 1);
1893 //    return 0;
1894 //}
1895
1896 // matchstar: search for c*regexp at beginning of text
1897 //int matchstar(int c, char* regexp, char* text)
1898 //{
1899 //    do
1900 //    {
1901 //        /* a * matches zero or more instances */
1902 //        if (matchhere(regexp, text))
1903 //            return 1;
1904 //    } while (*text != '\0' && (*text++ == c || c == '.'));
1905 //    return 0;
1906 //}
1907
1908 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
1909 //    ↳ long maximumGap)
1910 //{
1911 //    mininumGap = 0;
1912 //    maximumGap = 0;
1913 //    element = 0;
1914 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1915 //    {
1916 //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1917 //            mininumGap++;
1918 //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1919 //            maximumGap = long.MaxValue;
1920 //        else
1921 //            break;
1922 //    }
1923
1924 //    if (maximumGap < mininumGap)
1925 //        maximumGap = mininumGap;
1926 //}
1927
1928 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1929 private bool PatternMatchCore(LinkIndex element)
1930 {
1931     if (_patternPosition >= _pattern.Count)
1932     {
1933         _patternPosition = -2;
1934         return false;
1935     }
1936     var currentPatternBlock = _pattern[_patternPosition];
1937     if (currentPatternBlock.Type == PatternBlockType.Gap)
1938     {
1939         //var currentMatchingBlockLength = (_sequencePosition -
1940         //    ↳ _lastMatchedBlockPosition);
1941         if (_sequencePosition < currentPatternBlock.Start)
1942         {
1943             _sequencePosition++;
1944             return true; // Двигаемся дальше
1945         }
1946         // Это последний блок
1947         if (_pattern.Count == _patternPosition + 1)
1948         {
1949             _patternPosition++;
1950             _sequencePosition = 0;
1951             return false; // Полное соответствие
1952         }
1953     }
1954     else
1955     {
1956

```

```

1952         if (_sequencePosition > currentPatternBlock.Stop)
1953         {
1954             return false; // Соответствие невозможно
1955         }
1956         var nextPatternBlock = _pattern[_patternPosition + 1];
1957         if (_patternSequence[nextPatternBlock.Start] == element)
1958         {
1959             if (nextPatternBlock.Start < nextPatternBlock.Stop)
1960             {
1961                 _patternPosition++;
1962                 _sequencePosition = 1;
1963             }
1964             else
1965             {
1966                 _patternPosition += 2;
1967                 _sequencePosition = 0;
1968             }
1969         }
1970     }
1971 }
1972 else // currentPatternBlock.Type == PatternBlockType.Elements
1973 {
1974     var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1975     if (_patternSequence[patternElementPosition] != element)
1976     {
1977         return false; // Соответствие невозможно
1978     }
1979     if (patternElementPosition == currentPatternBlock.Stop)
1980     {
1981         _patternPosition++;
1982         _sequencePosition = 0;
1983     }
1984     else
1985     {
1986         _sequencePosition++;
1987     }
1988 }
1989 return true;
1990 //if (_patternSequence[_patternPosition] != element)
1991 //    return false;
1992 //else
1993 //{
1994 //    _sequencePosition++;
1995 //    _patternPosition++;
1996 //    return true;
1997 //}
1998 //////////////////////////////////////////////////
1999 //if (_filterPosition == _patternSequence.Length)
2000 //{
2001 //    _filterPosition = -2; // Длиннее чем нужно
2002 //    return false;
2003 //}
2004 //if (element != _patternSequence[_filterPosition])
2005 //{
2006 //    _filterPosition = -1;
2007 //    return false; // Начинается иначе
2008 //}
2009 //_filterPosition++;
2010 //if (_filterPosition == (_patternSequence.Length - 1))
2011 //    return false;
2012 //if (_filterPosition >= 0)
2013 //{
2014 //    if (element == _patternSequence[_filterPosition + 1])
2015 //        _filterPosition++;
2016 //    else
2017 //        return false;
2018 //}
2019 //if (_filterPosition < 0)
2020 //{
2021 //    if (element == _patternSequence[0])
2022 //        _filterPosition = 0;
2023 //}
2024 }
2025
2026 [MethodImpl(MethodImplOptions.AggressiveInlining)]
2027 public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
2028 {
2029     foreach (var sequenceToMatch in sequencesToMatch)
2030     {

```

```

2031         if (PatternMatch(sequenceToMatch))
2032         {
2033             _results.Add(sequenceToMatch);
2034         }
2035     }
2036 }
2037 }
2038
2039 #endregion
2040 }
2041 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections;
6  using Platform.Collections.Lists;
7  using Platform.Collections.Stacks;
8  using Platform.Threading.Synchronization;
9  using Platform.Data.Doublets.Sequences.Walkers;
10 using LinkIndex = System.UInt64;
11
12 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     /// <summary>
17     /// Представляет коллекцию последовательностей связей.
18     /// </summary>
19     /// <remarks>
20     /// Обязательно реализовать атомарность каждого публичного метода.
21     ///
22     /// TODO:
23     ///
24     /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
25     /// через естественную группировку по unicode типам, все whitespace вместе, все символы
26     /// вместе, все числа вместе и т.п.
27     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
28     /// графа)
29     ///
30     /// х*у - найти все связи между, в последовательностях любой формы, если не стоит
31     /// ограничитель на то, что является последовательностью, а что нет,
32     /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
33     /// порядке.
34     ///
35     /// Рост последовательности слева и справа.
36     /// Поиск со звёздочкой.
37     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
38     /// так же проблема может быть решена при реализации дистанционных триггеров.
39     /// Нужны ли уникальные указатели вообще?
40     /// Что если обращение к информации будет происходить через содержимое всегда?
41     ///
42     /// Писать тесты.
43     ///
44     /// Можно убрать зависимость от конкретной реализации Links,
45     /// на зависимость от абстрактного элемента, который может быть представлен несколькими
46     /// способами.
47     ///
48     /// Можно ли как-то сделать один общий интерфейс
49     ///
50     /// Блокчейн и/или гит для распределённой записи транзакций.
51     ///
52     /// </remarks>
53     public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
54     {
55         (после завершения реализации Sequences)
56
57         /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
58         /// связей.</summary>
59         public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
60
61         public SequencesOptions<LinkIndex> Options { get; }
62         public SynchronizedLinks<LinkIndex> Links { get; }
63         private readonly ISynchronization _sync;
64
65         public LinksConstants<LinkIndex> Constants { get; }

```



```

60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
62 {
63     Links = links;
64     _sync = links.SyncRoot;
65     Options = options;
66     Options.ValidateOptions();
67     Options.InitOptions(Links);
68     Constants = links.Constants;
69 }
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
73     ↳ SequencesOptions<LinkIndex>()) { }
74
75 [MethodImpl(MethodImplOptions.AggressiveInlining)]
76 public bool IsSequence(LinkIndex sequence)
77 {
78     return _sync.ExecuteReadOperation(() =>
79     {
80         if (Options.UseSequenceMarker)
81         {
82             return Options.MarkedSequenceMatcher.IsMatched(sequence);
83         }
84         return !Links.Unsync.IsPartialPoint(sequence);
85     });
86 }
87
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 private LinkIndex GetSequenceByElements(LinkIndex sequence)
90 {
91     if (Options.UseSequenceMarker)
92     {
93         return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
94     }
95     return sequence;
96 }
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 private LinkIndex GetSequenceElements(LinkIndex sequence)
100 {
101     if (Options.UseSequenceMarker)
102     {
103         var linkContents = new Link<ulong>(Links.GetLink(sequence));
104         if (linkContents.Source == Options.SequenceMarkerLink)
105         {
106             return linkContents.Target;
107         }
108         if (linkContents.Target == Options.SequenceMarkerLink)
109         {
110             return linkContents.Source;
111         }
112     }
113     return sequence;
114 }
115
116 #region Count
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 public LinkIndex Count(ICollection<LinkIndex> restrictions)
120 {
121     if (restrictions.IsNullOrEmpty())
122     {
123         return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
124     }
125     if (restrictions.Count == 1) // Первая связь это адрес
126     {
127         var sequenceIndex = restrictions[0];
128         if (sequenceIndex == Constants.Null)
129         {
130             return 0;
131         }
132         if (sequenceIndex == Constants.Any)
133         {
134             return Count(null);
135         }
136         if (Options.UseSequenceMarker)
137         {

```

```

138         return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
139     }
140     return Links.Exists(sequenceIndex) ? 1UL : 0;
141 }
142 throw new NotImplementedException();
143 }
144
145 [MethodImpl(MethodImplOptions.AggressiveInlining)]
146 private LinkIndex CountUsages(params LinkIndex[] restrictions)
147 {
148     if (restrictions.Length == 0)
149     {
150         return 0;
151     }
152     if (restrictions.Length == 1) // Первая связь это адрес
153     {
154         if (restrictions[0] == Constants.Null)
155         {
156             return 0;
157         }
158         var any = Constants.Any;
159         if (Options.UseSequenceMarker)
160         {
161             var elementsLink = GetSequenceElements(restrictions[0]);
162             var sequenceLink = GetSequenceByElements(elementsLink);
163             if (sequenceLink != Constants.Null)
164             {
165                 return Links.Count(any, sequenceLink) + Links.Count(any, elementsLink) -
166                     ↪ 1;
167             }
168             return Links.Count(any, elementsLink);
169         }
170         return Links.Count(any, restrictions[0]);
171     }
172     throw new NotImplementedException();
173 }
174
175 #endregion
176
177 #region Create
178
179 [MethodImpl(MethodImplOptions.AggressiveInlining)]
180 public LinkIndex Create(ICollection<LinkIndex> restrictions)
181 {
182     return _sync.ExecuteWriteOperation(() =>
183     {
184         if (restrictions.IsNullOrEmpty())
185         {
186             return Constants.Null;
187         }
188         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
189         return CreateCore(restrictions);
190     });
191 }
192
193 [MethodImpl(MethodImplOptions.AggressiveInlining)]
194 private LinkIndex CreateCore(ICollection<LinkIndex> restrictions)
195 {
196     LinkIndex[] sequence = restrictions.SkipFirst();
197     if (Options.UseIndex)
198     {
199         Options.Index.Add(sequence);
200     }
201     var sequenceRoot = default(LinkIndex);
202     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
203     {
204         var matches = Each(restrictions);
205         if (matches.Count > 0)
206         {
207             sequenceRoot = matches[0];
208         }
209     }
210     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
211     {
212         return CompactCore(sequence);
213     }
214     if (sequenceRoot == default)
215     {

```

```

sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
}
if (Options.UseSequenceMarker)
{
    return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
}
return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
}

#endregion

#region Each

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public List<LinkIndex> Each(IList<LinkIndex> sequence)
{
    var results = new List<LinkIndex>();
    var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
    Each(filler.AddFirstAndReturnConstant, sequence);
    return results;
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
    ↪ restrictions)
{
    return _sync.ExecuteReadOperation(() =>
    {
        if (restrictions.IsNullOrEmpty())
        {
            return Constants.Continue;
        }
        Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
        if (restrictions.Count == 1)
        {
            var link = restrictions[0];
            var any = Constants.Any;
            if (link == any)
            {
                if (Options.UseSequenceMarker)
                {
                    return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
                        ↪ Options.SequenceMarkerLink, any));
                }
                else
                {
                    return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
                        ↪ any));
                }
            }
            if (Options.UseSequenceMarker)
            {
                var sequenceLinkValues = Links.Unsync.GetLink(link);
                if (sequenceLinkValues[Constants.SourcePart] ==
                    ↪ Options.SequenceMarkerLink)
                {
                    link = sequenceLinkValues[Constants.TargetPart];
                }
            }
            var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
            sequence[0] = link;
            return handler(sequence);
        }
        else if (restrictions.Count == 2)
        {
            throw new NotImplementedException();
        }
        else if (restrictions.Count == 3)
        {
            return Links.Unsync.Each(handler, restrictions);
        }
        else
        {
            var sequence = restrictions.SkipFirst();
            if (Options.UseIndex && !Options.Index.MightContain(sequence))
            {
                return Constants.Break;
            }
        }
    });
}

```

```

289         return EachCore(handler, sequence);
290     }
291 });
292 }
293
294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
    ↪ values)
296 {
297     var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
298     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
    ↪ Id.
299     Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
    ↪ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
    ↪ matcher.HandleFullMatched;
300     //if (sequence.Length >= 2)
301     if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
302     {
303         return Constants.Break;
304     }
305     var last = values.Count - 2;
306     for (var i = 1; i < last; i++)
307     {
308         if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
    ↪ Constants.Continue)
309         {
310             return Constants.Break;
311         }
312     }
313     if (values.Count >= 3)
314     {
315         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
    ↪ != Constants.Continue)
316         {
317             return Constants.Break;
318         }
319     }
320     return Constants.Continue;
321 }
322
323 [MethodImpl(MethodImplOptions.AggressiveInlining)]
324 private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↪ left, LinkIndex right)
325 {
326     return Links.Unsync.Each(doublet =>
327     {
328         var doubletIndex = doublet[Constants.IndexPart];
329         if (StepRight(handler, doubletIndex, right) != Constants.Continue)
330         {
331             return Constants.Break;
332         }
333         if (left != doubletIndex)
334         {
335             return PartialStepRight(handler, doubletIndex, right);
336         }
337         return Constants.Continue;
338     }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
339 }
340
341 [MethodImpl(MethodImplOptions.AggressiveInlining)]
342 private LinkIndex StepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↪ LinkIndex right) => Links.Unsync.Each(rightStep => TryStepRightUp(handler, right,
    ↪ rightStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, left,
    ↪ Constants.Any));
343
344 [MethodImpl(MethodImplOptions.AggressiveInlining)]
345 private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↪ right, LinkIndex stepFrom)
346 {
347     var upStep = stepFrom;
348     var firstSource = Links.Unsync.GetTarget(upStep);
349     while (firstSource != right && firstSource != upStep)
350     {
351         upStep = firstSource;
352         firstSource = Links.Unsync.GetSource(upStep);
353     }
354     if (firstSource == right)
355     {

```

```

356         return handler(new LinkAddress<LinkIndex>(stepFrom));
357     }
358     return Constants.Continue;
359 }
360
361 [MethodImpl(MethodImplOptions.AggressiveInlining)]
362 private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↳ LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
    ↳ leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
    ↳ right));
363
364 [MethodImpl(MethodImplOptions.AggressiveInlining)]
365 private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↳ left, LinkIndex stepFrom)
366 {
367     var upStep = stepFrom;
368     var firstTarget = Links.Unsync.GetSource(upStep);
369     while (firstTarget != left && firstTarget != upStep)
370     {
371         upStep = firstTarget;
372         firstTarget = Links.Unsync.GetTarget(upStep);
373     }
374     if (firstTarget == left)
375     {
376         return handler(new LinkAddress<LinkIndex>(stepFrom));
377     }
378     return Constants.Continue;
379 }
380
381 #endregion
382
383 #region Update
384
385 [MethodImpl(MethodImplOptions.AggressiveInlining)]
386 public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
387 {
388     var sequence = restrictions.SkipFirst();
389     var newSequence = substitution.SkipFirst();
390     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
391     {
392         return Constants.Null;
393     }
394     if (sequence.IsNullOrEmpty())
395     {
396         return Create(substitution);
397     }
398     if (newSequence.IsNullOrEmpty())
399     {
400         Delete(restrictions);
401         return Constants.Null;
402     }
403     return _sync.ExecuteWriteOperation((Func<ulong>)(() =>
404     {
405         ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
406         Links.EnsureLinkExists(newSequence);
407         return UpdateCore(sequence, newSequence);
408     })));
409 }
410
411 [MethodImpl(MethodImplOptions.AggressiveInlining)]
412 private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence)
413 {
414     LinkIndex bestVariant;
415     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
    ↳ !sequence.EqualTo(newSequence))
416     {
417         bestVariant = CompactCore(newSequence);
418     }
419     else
420     {
421         bestVariant = CreateCore(newSequence);
422     }
423     // TODO: Check all options only ones before loop execution
424     // Возможно нужно две версии Each, возвращающий фактические последовательности и с
    ↳ маркером,
425     // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
    ↳ можно получить имея только фактические последовательности.
426     foreach (var variant in Each(sequence))
427     {

```

```

428         if (variant != bestVariant)
429         {
430             UpdateOneCore(variant, bestVariant);
431         }
432     }
433     return bestVariant;
434 }
435
436 [MethodImpl(MethodImplOptions.AggressiveInlining)]
437 private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
438 {
439     if (Options.UseGarbageCollection)
440     {
441         var sequenceElements = GetSequenceElements(sequence);
442         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
443         var sequenceLink = GetSequenceByElements(sequenceElements);
444         var newSequenceElements = GetSequenceElements(newSequence);
445         var newSequenceLink = GetSequenceByElements(newSequenceElements);
446         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
447         {
448             if (sequenceLink != Constants.Null)
449             {
450                 Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
451             }
452             Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
453         }
454         ClearGarbage(sequenceElementsContents.Source);
455         ClearGarbage(sequenceElementsContents.Target);
456     }
457     else
458     {
459         if (Options.UseSequenceMarker)
460         {
461             var sequenceElements = GetSequenceElements(sequence);
462             var sequenceLink = GetSequenceByElements(sequenceElements);
463             var newSequenceElements = GetSequenceElements(newSequence);
464             var newSequenceLink = GetSequenceByElements(newSequenceElements);
465             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
466             {
467                 if (sequenceLink != Constants.Null)
468                 {
469                     Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
470                 }
471                 Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
472             }
473         }
474         else
475         {
476             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
477             {
478                 Links.Unsync.MergeAndDelete(sequence, newSequence);
479             }
480         }
481     }
482 }
483
484 #endregion
485
486 #region Delete
487
488 [MethodImpl(MethodImplOptions.AggressiveInlining)]
489 public void Delete(IList<LinkIndex> restrictions)
490 {
491     _sync.ExecuteWriteOperation(() =>
492     {
493         var sequence = restrictions.SkipFirst();
494         // TODO: Check all options only ones before loop execution
495         foreach (var linkToDelete in Each(sequence))
496         {
497             DeleteOneCore(linkToDelete);
498         }
499     });
500 }
501
502 [MethodImpl(MethodImplOptions.AggressiveInlining)]
503 private void DeleteOneCore(LinkIndex link)
504 {
505     if (Options.UseGarbageCollection)

```

```

506 {
507     var sequenceElements = GetSequenceElements(link);
508     var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
509     var sequenceLink = GetSequenceByElements(sequenceElements);
510     if (Options.UseCascadeDelete || CountUsages(link) == 0)
511     {
512         if (sequenceLink != Constants.Null)
513         {
514             Links.Unsync.Delete(sequenceLink);
515         }
516         Links.Unsync.Delete(link);
517     }
518     ClearGarbage(sequenceElementsContents.Source);
519     ClearGarbage(sequenceElementsContents.Target);
520 }
521 else
522 {
523     if (Options.UseSequenceMarker)
524     {
525         var sequenceElements = GetSequenceElements(link);
526         var sequenceLink = GetSequenceByElements(sequenceElements);
527         if (Options.UseCascadeDelete || CountUsages(link) == 0)
528         {
529             if (sequenceLink != Constants.Null)
530             {
531                 Links.Unsync.Delete(sequenceLink);
532             }
533             Links.Unsync.Delete(link);
534         }
535     }
536     else
537     {
538         if (Options.UseCascadeDelete || CountUsages(link) == 0)
539         {
540             Links.Unsync.Delete(link);
541         }
542     }
543 }
544 }
545
546 #endregion
547
548 #region Compactification
549
550 [MethodImpl(MethodImplOptions.AggressiveInlining)]
551 public void CompactAll()
552 {
553     _sync.ExecuteWriteOperation(() =>
554     {
555         var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
556         for (int i = 0; i < sequences.Count; i++)
557         {
558             var sequence = this.ToList(sequences[i]);
559             Compact(sequence.ShiftRight());
560         }
561     });
562 }
563
564 /// <remarks>
565 /// bestVariant можно выбирать по максимальному числу использований,
566 /// но балансированный позволяет гарантировать уникальность (если есть возможность,
567 /// гарантировать его использование в других местах).
568 ///
569 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
570 /// </remarks>
571 [MethodImpl(MethodImplOptions.AggressiveInlining)]
572 public LinkIndex Compact(ICollection<LinkIndex> sequence)
573 {
574     return _sync.ExecuteWriteOperation(() =>
575     {
576         if (sequence.IsNullOrEmpty())
577         {
578             return Constants.Null;
579         }
580         Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
581         return CompactCore(sequence);
582     });
583 }

```

```

584 [MethodImpl(MethodImplOptions.AggressiveInlining)]
585 private LinkIndex CompactCore(ICollection<LinkIndex> sequence) => UpdateCore(sequence,
586     ↪ sequence);
587
588 #endregion
589
590 #region Garbage Collection
591
592 /// <remarks>
593 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
594     ↪ определить извне или в унаследованном классе
595 /// </remarks>
596 [MethodImpl(MethodImplOptions.AggressiveInlining)]
597 private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
598     ↪ !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
599
600 [MethodImpl(MethodImplOptions.AggressiveInlining)]
601 private void ClearGarbage(LinkIndex link)
602 {
603     if (IsGarbage(link))
604     {
605         var contents = new Link<ulong>(Links.GetLink(link));
606         Links.Unsync.Delete(link);
607         ClearGarbage(contents.Source);
608         ClearGarbage(contents.Target);
609     }
610 }
611
612 #endregion
613
614 #region Walkers
615
616 [MethodImpl(MethodImplOptions.AggressiveInlining)]
617 public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
618 {
619     return _sync.ExecuteReadOperation(() =>
620     {
621         var links = Links.Unsync;
622         foreach (var part in Options.Walker.Walk(sequence))
623         {
624             if (!handler(part))
625             {
626                 return false;
627             }
628         }
629         return true;
630     });
631 }
632
633 public class Matcher : RightSequenceWalker<LinkIndex>
634 {
635     private readonly Sequences _sequences;
636     private readonly ICollection<LinkIndex> _patternSequence;
637     private readonly HashSet<LinkIndex> _linksInSequence;
638     private readonly HashSet<LinkIndex> _results;
639     private readonly Func<ICollection<LinkIndex>, LinkIndex> _stopableHandler;
640     private readonly HashSet<LinkIndex> _readAsElements;
641     private int _filterPosition;
642
643     [MethodImpl(MethodImplOptions.AggressiveInlining)]
644     public Matcher(Sequences sequences, ICollection<LinkIndex> patternSequence,
645         ↪ HashSet<LinkIndex> results, Func<ICollection<LinkIndex>, LinkIndex> stopableHandler,
646         ↪ HashSet<LinkIndex> readAsElements = null)
647         : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
648     {
649         _sequences = sequences;
650         _patternSequence = patternSequence;
651         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
652             ↪ _links.Constants.Any && x != ZeroOrMany));
653         _results = results;
654         _stopableHandler = stopableHandler;
655         _readAsElements = readAsElements;
656     }
657
658     [MethodImpl(MethodImplOptions.AggressiveInlining)]
659     protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
660         ↪ (_readAsElements != null && _readAsElements.Contains(link)) ||
661         ↪ _linksInSequence.Contains(link);

```



```

656 [MethodImpl(MethodImplOptions.AggressiveInlining)]
657 public bool FullMatch(LinkIndex sequenceToMatch)
658 {
659     _filterPosition = 0;
660     foreach (var part in Walk(sequenceToMatch))
661     {
662         if (!FullMatchCore(part))
663         {
664             break;
665         }
666     }
667     return _filterPosition == _patternSequence.Count;
668 }
669
670 [MethodImpl(MethodImplOptions.AggressiveInlining)]
671 private bool FullMatchCore(LinkIndex element)
672 {
673     if (_filterPosition == _patternSequence.Count)
674     {
675         _filterPosition = -2; // Длиннее чем нужно
676         return false;
677     }
678     if (_patternSequence[_filterPosition] != _links.Constants.Any
679         && element != _patternSequence[_filterPosition])
680     {
681         _filterPosition = -1;
682         return false; // Начинается/Продолжается иначе
683     }
684     _filterPosition++;
685     return true;
686 }
687
688 [MethodImpl(MethodImplOptions.AggressiveInlining)]
689 public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
690 {
691     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
692     if (FullMatch(sequenceToMatch))
693     {
694         _results.Add(sequenceToMatch);
695     }
696 }
697
698 [MethodImpl(MethodImplOptions.AggressiveInlining)]
699 public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
700 {
701     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
702     if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
703     {
704         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
705     }
706     return _links.Constants.Continue;
707 }
708
709 [MethodImpl(MethodImplOptions.AggressiveInlining)]
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) &&
715         ↪ _results.Add(sequenceToMatch))
716     {
717         return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
718     }
719     return _links.Constants.Continue;
720 }
721
722 /// <remarks>
723 /// TODO: Add support for LinksConstants.Any
724 /// </remarks>
725 [MethodImpl(MethodImplOptions.AggressiveInlining)]
726 public bool PartialMatch(LinkIndex sequenceToMatch)
727 {
728     _filterPosition = -1;
729     foreach (var part in Walk(sequenceToMatch))
730     {
731         if (!PartialMatchCore(part))
732         {
733             break;
734         }
735     }

```

```

734     }
735     return _filterPosition == _patternSequence.Count - 1;
736 }
737
738 [MethodImpl(MethodImplOptions.AggressiveInlining)]
739 private bool PartialMatchCore(LinkIndex element)
740 {
741     if (_filterPosition == (_patternSequence.Count - 1))
742     {
743         return false; // Нашлось
744     }
745     if (_filterPosition >= 0)
746     {
747         if (element == _patternSequence[_filterPosition + 1])
748         {
749             _filterPosition++;
750         }
751         else
752         {
753             _filterPosition = -1;
754         }
755     }
756     if (_filterPosition < 0)
757     {
758         if (element == _patternSequence[0])
759         {
760             _filterPosition = 0;
761         }
762     }
763     return true; // Ищем дальше
764 }
765
766 [MethodImpl(MethodImplOptions.AggressiveInlining)]
767 public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
768 {
769     if (PartialMatch(sequenceToMatch))
770     {
771         _results.Add(sequenceToMatch);
772     }
773 }
774
775 [MethodImpl(MethodImplOptions.AggressiveInlining)]
776 public LinkIndex HandlePartialMatched(ICollection<LinkIndex> restrictions)
777 {
778     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
779     if (PartialMatch(sequenceToMatch))
780     {
781         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
782     }
783     return _links.Constants.Continue;
784 }
785
786 [MethodImpl(MethodImplOptions.AggressiveInlining)]
787 public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)
788 {
789     foreach (var sequenceToMatch in sequencesToMatch)
790     {
791         if (PartialMatch(sequenceToMatch))
792         {
793             _results.Add(sequenceToMatch);
794         }
795     }
796 }
797
798 [MethodImpl(MethodImplOptions.AggressiveInlining)]
799 public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
800 ↪ sequencesToMatch)
801 {
802     foreach (var sequenceToMatch in sequencesToMatch)
803     {
804         if (PartialMatch(sequenceToMatch))
805         {
806             _readAsElements.Add(sequenceToMatch);
807             _results.Add(sequenceToMatch);
808         }
809     }
810 }

```

```

811
812     #endregion
813 }
814 }

```

1.117 ./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 }

```

1.118 ./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 }

```

```

36     }
37
38     public bool UseCascadeDelete
39     {
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         get;
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         set;
44     }
45
46     public bool UseIndex
47     {
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         get;
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         set;
52     } // TODO: Update Index on sequence update/delete.
53
54     public bool UseSequenceMarker
55     {
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         get;
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         set;
60     }
61
62     public bool UseCompression
63     {
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         get;
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         set;
68     }
69
70     public bool UseGarbageCollection
71     {
72         [MethodImpl(MethodImplOptions.AggressiveInlining)]
73         get;
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]
75         set;
76     }
77
78     public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting
79     {
80         [MethodImpl(MethodImplOptions.AggressiveInlining)]
81         get;
82         [MethodImpl(MethodImplOptions.AggressiveInlining)]
83         set;
84     }
85
86     public bool EnforceSingleSequenceVersionOnWriteBasedOnNew
87     {
88         [MethodImpl(MethodImplOptions.AggressiveInlining)]
89         get;
90         [MethodImpl(MethodImplOptions.AggressiveInlining)]
91         set;
92     }
93
94     public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher
95     {
96         [MethodImpl(MethodImplOptions.AggressiveInlining)]
97         get;
98         [MethodImpl(MethodImplOptions.AggressiveInlining)]
99         set;
100    }
101
102    public IConverter<IList<TLink>, TLink> LinksToSequenceConverter
103    {
104        [MethodImpl(MethodImplOptions.AggressiveInlining)]
105        get;
106        [MethodImpl(MethodImplOptions.AggressiveInlining)]
107        set;
108    }
109
110    public ISequenceIndex<TLink> Index
111    {
112        [MethodImpl(MethodImplOptions.AggressiveInlining)]
113        get;
114        [MethodImpl(MethodImplOptions.AggressiveInlining)]
115        set;
116    }

```

```

117 public ISequenceWalker<TLink> Walker
118 {
119     [MethodImpl(MethodImplOptions.AggressiveInlining)]
120     get;
121     [MethodImpl(MethodImplOptions.AggressiveInlining)]
122     set;
123 }
124
125 public bool ReadFullSequence
126 {
127     [MethodImpl(MethodImplOptions.AggressiveInlining)]
128     get;
129     [MethodImpl(MethodImplOptions.AggressiveInlining)]
130     set;
131 }
132
133 // TODO: Реализовать компактификацию при чтении
134 //public bool EnforceSingleSequenceVersionOnRead { get; set; }
135 //public bool UseRequestMarker { get; set; }
136 //public bool StoreRequestResults { get; set; }
137
138 [MethodImpl(MethodImplOptions.AggressiveInlining)]
139 public void InitOptions(ISynchronizedLinks<TLink> links)
140 {
141     if (UseSequenceMarker)
142     {
143         if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
144         {
145             SequenceMarkerLink = links.CreatePoint();
146         }
147         else
148         {
149             if (!links.Exists(SequenceMarkerLink))
150             {
151                 var link = links.CreatePoint();
152                 if (!_equalityComparer.Equals(link, SequenceMarkerLink))
153                 {
154                     throw new InvalidOperationException("Cannot recreate sequence marker
155                     ↪ link.");
156                 }
157             }
158         }
159         if (MarkedSequenceMatcher == null)
160         {
161             MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
162             ↪ SequenceMarkerLink);
163         }
164         var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
165         if (UseCompression)
166         {
167             if (LinksToSequenceConverter == null)
168             {
169                 ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
170                 if (UseSequenceMarker)
171                 {
172                     totalSequenceSymbolFrequencyCounter = new
173                     ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
174                     ↪ MarkedSequenceMatcher);
175                 }
176                 else
177                 {
178                     totalSequenceSymbolFrequencyCounter = new
179                     ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
180                 }
181                 var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
182                 ↪ totalSequenceSymbolFrequencyCounter);
183                 var compressingConverter = new CompressingConverter<TLink>(links,
184                 ↪ balancedVariantConverter, doubletFrequenciesCache);
185                 LinksToSequenceConverter = compressingConverter;
186             }
187         }
188     }
189     else
190     {
191         if (LinksToSequenceConverter == null)
192         {
193             LinksToSequenceConverter = balancedVariantConverter;
194         }
195     }
196 }

```

```

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
206         ↪ option must be on.");
207     }
208 }
209 }

```

1.119 ./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 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Collections.Stacks;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Sequences.Walkers
9 {
10     public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
14         ↪ isElement) : base(links, stack, isElement) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
18         ↪ links.IsPartialPoint) { }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override TLink GetNextElementAfterPop(TLink element) =>
22         ↪ _links.GetSource(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override TLink GetNextElementAfterPush(TLink element) =>
26         ↪ _links.GetTarget(element);
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override IEnumerable<TLink> WalkContents(TLink element)
30         {
31             var links = _links;
32             var parts = links.GetLink(element);
33             var start = links.Constants.SourcePart;
34             for (var i = parts.Count - 1; i >= start; i--)
35             {
36                 var part = parts[i];
37                 if (IsElement(part))
38                 {
39                     yield return part;
40                 }
41             }
42         }
43     }
44 }

```

```

36     }
37 }
38 }
39 }
40 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  //#define USEARRAYPOOL
8  #if USEARRAYPOOL
9  using Platform.Collections;
10 #endif
11
12 namespace Platform.Data.Doublets.Sequences.Walkers
13 {
14     public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
15     {
16         private static readonly EqualityComparer<TLink> _equalityComparer =
17             ↳ EqualityComparer<TLink>.Default;
18
19         private readonly Func<TLink, bool> _isElement;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
23             ↳ base(links) => _isElement = isElement;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
27             ↳ _links.IsPartialPoint;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public TLink[] ToArray(TLink sequence)
34         {
35             var length = 1;
36             var array = new TLink[length];
37             array[0] = sequence;
38             if (_isElement(sequence))
39             {
40                 return array;
41             }
42             bool hasElements;
43             do
44             {
45                 length *= 2;
46                 #if USEARRAYPOOL
47                     var nextArray = ArrayPool.Allocate<ulong>(length);
48                 #else
49                     var nextArray = new TLink[length];
50                 #endif
51                 hasElements = false;
52                 for (var i = 0; i < array.Length; i++)
53                 {
54                     var candidate = array[i];
55                     if (_equalityComparer.Equals(array[i], default))
56                     {
57                         continue;
58                     }
59                     var doubletOffset = i * 2;
60                     if (_isElement(candidate))
61                     {
62                         nextArray[doubletOffset] = candidate;
63                     }
64                     else
65                     {
66                         var links = _links;
67                         var link = links.GetLink(candidate);
68                         var linkSource = links.GetSource(link);
69                         var linkTarget = links.GetTarget(link);
70                         nextArray[doubletOffset] = linkSource;
71                         nextArray[doubletOffset + 1] = linkTarget;
72                     }
73                     if (!hasElements)
74                     {
75

```

```

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 }

```

1.122 ./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)]

```



```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Stacks;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.Walkers
9  {
10     public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
11         ↪ ISequenceWalker<TLink>
12     {
13         private readonly IStack<TLink> _stack;
14         private readonly Func<TLink, bool> _isElement;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
18             ↪ isElement) : base(links)
19         {
20             _stack = stack;
21             _isElement = isElement;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
26             ↪ stack, links.IsPartialPoint) { }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public IEnumerable<TLink> Walk(TLink sequence)
30         {
31             _stack.Clear();
32             var element = sequence;
33             if (IsElement(element))
34             {
35                 yield return element;
36             }
37             else
38             {
39                 while (true)
40                 {
41                     if (IsElement(element))
42                     {
43                         if (_stack.IsEmpty)
44                         {
45                             break;
46                         }
47                         element = _stack.Pop();
48                         foreach (var output in WalkContents(element))
49                         {
50                             yield return output;
51                         }
52                         element = GetNextElementAfterPop(element);
53                     }
54                     else
55                     {

```

```

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 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Collections.Stacks;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Stacks
8  {
9      public class Stack<TLink> : LinksOperatorBase<TLink>, IStack<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13
14         private readonly TLink _stack;
15
16         public bool IsEmpty
17         {
18             [MethodImpl(MethodImplOptions.AggressiveInlining)]
19             get => _equalityComparer.Equals(Peek(), _stack);
20         }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public Stack(ILinks<TLink> links, TLink stack) : base(links) => _stack = stack;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         private TLink GetStackMarker() => _links.GetSource(_stack);
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         private TLink GetTop() => _links.GetTarget(_stack);
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public TLink Peek() => _links.GetTarget(GetTop());
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public TLink Pop()
36         {
37             var element = Peek();
38             if (!_equalityComparer.Equals(element, _stack))
39             {
40                 var top = GetTop();
41                 var previousTop = _links.GetSource(top);
42                 _links.Update(_stack, GetStackMarker(), previousTop);
43                 _links.Delete(top);
44             }
45             return element;
46         }
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
50             ↪ _links.GetOrCreate(GetTop(), element));
51     }
52 }

```

1.125 ./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 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Data.Doublets;
5 using Platform.Threading.Synchronization;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets
10 {
11     /// <remarks>
12     /// TODO: Autogeneration of synchronized wrapper (decorator).
13     /// TODO: Try to unfold code of each method using IL generation for performance improvements.
14     /// TODO: Or even to unfold multiple layers of implementations.
15     /// </remarks>
16     public class SynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress>
17     {
18         public LinksConstants<TLinkAddress> Constants
19         {
20             [MethodImpl(MethodImplOptions.AggressiveInlining)]
21             get;
22         }
23
24         public ISynchronization SyncRoot
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28         }
29
30         public ILinks<TLinkAddress> Sync
31         {
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             get;
34         }
35
36         public ILinks<TLinkAddress> Unsync
37         {
38             [MethodImpl(MethodImplOptions.AggressiveInlining)]
39             get;
40         }
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
44             ↳ ReaderWriterLockSynchronization(), links) { }
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
48         {
49             SyncRoot = synchronization;
50             Sync = this;
51             Unsync = links;
52             Constants = links.Constants;
53         }
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public TLinkAddress Count(IList<TLinkAddress> restriction) =>
57             ↳ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
61             ↳ IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
62             ↳ restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

61     public TLinkAddress Create(ICollection<TLinkAddress> restrictions) =>
        ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     public TLinkAddress Update(ICollection<TLinkAddress> restrictions, ICollection<TLinkAddress>
        ↳ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
        ↳ Unsync.Update);
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     public void Delete(ICollection<TLinkAddress> restrictions) =>
        ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
68
69     //public T Trigger(ICollection<T> restriction, Func<ICollection<T>, ICollection<T>, T> matchedHandler,
        ↳ ICollection<T> substitution, Func<ICollection<T>, ICollection<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 }

```

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

```

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

```

```

47 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
    ↳ Func<Link<ulong>, bool> isElement, Action<StringBuilder, Link<ulong>> appendElement,
    ↳ bool renderIndex = false, bool renderDebug = false)
48 {
49     var sb = new StringBuilder();
50     var visited = new HashSet<ulong>();
51     links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
    ↳ renderDebug);
52     return sb.ToString();
53 }
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
    ↳ HashSet<ulong> visited, ulong linkIndex, Func<Link<ulong>, bool> isElement,
    ↳ Action<StringBuilder, Link<ulong>> appendElement, bool renderIndex = false, bool
    ↳ renderDebug = false)
57 {
58     if (sb == null)
59     {
60         throw new ArgumentNullException(nameof(sb));
61     }
62     if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
    ↳ Constants.Itself)
63     {
64         return;
65     }
66     if (links.Exists(linkIndex))
67     {
68         if (visited.Add(linkIndex))
69         {
70             sb.Append('(');
71             var link = new Link<ulong>(links.GetLink(linkIndex));
72             if (renderIndex)
73             {
74                 sb.Append(link.Index);
75                 sb.Append(':');
76             }
77             if (link.Source == link.Index)
78             {
79                 sb.Append(link.Index);
80             }
81             else
82             {
83                 var source = new Link<ulong>(links.GetLink(link.Source));
84                 if (isElement(source))
85                 {
86                     appendElement(sb, source);
87                 }
88                 else
89                 {
90                     links.AppendStructure(sb, visited, source.Index, isElement,
    ↳ appendElement, renderIndex);
91                 }
92             }
93             sb.Append(' ');
94             if (link.Target == link.Index)
95             {
96                 sb.Append(link.Index);
97             }
98             else
99             {
100                 var target = new Link<ulong>(links.GetLink(link.Target));
101                 if (isElement(target))
102                 {
103                     appendElement(sb, target);
104                 }
105                 else
106                 {
107                     links.AppendStructure(sb, visited, target.Index, isElement,
    ↳ appendElement, renderIndex);
108                 }
109             }
110             sb.Append(')');
111         }
112     }
113     else
114     {
115         if (renderDebug)
116         {

```

```

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 }

```

1.128 ./csharp/Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using System.IO;
5  using System.Runtime.CompilerServices;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Platform.Disposables;
9  using Platform.Timestamps;
10 using Platform.Unsafe;
11 using Platform.IO;
12 using Platform.Data.Doublets.Decorators;
13 using Platform.Exceptions;
14
15 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
16
17 namespace Platform.Data.Doublets
18 {
19     public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
20     {
21         /// <remarks>
22         /// Альтернативные варианты хранения трансформации (элемента транзакции):
23         ///
24         /// private enum TransitionType
25         /// {
26         ///     Creation,
27         ///     UpdateOf,
28         ///     UpdateTo,
29         ///     Deletion
30         /// }
31         ///
32         /// private struct Transition
33         /// {
34         ///     public ulong TransactionId;
35         ///     public UniqueTimestamp Timestamp;
36         ///     public TransactionItemType Type;
37         ///     public Link Source;
38         ///     public Link Linker;
39         ///     public Link Target;
40         /// }
41         /// Или
42         ///
43         /// public struct TransitionHeader
44         /// {
45         ///     public ulong TransactionIdCombined;
46         ///     public ulong TimestampCombined;
47         ///
48         ///     public ulong TransactionId
49         ///     {
50         ///         get
51         ///         {
52         ///             return (ulong) mask & TransactionIdCombined;
53         ///         }
54         ///     }
55         ///
56         ///     public UniqueTimestamp Timestamp
57         ///     {
58         ///         get
59         ///         {
60         ///             return (UniqueTimestamp)mask & TransactionIdCombined;

```

```

62     ///     }
63     /// }
64     ///
65     public TransactionItemType Type
66     {
67         ///     get
68         ///     {
69             ///         // Использовать по одному биту из TransactionId и Timestamp,
70             ///         // для значения в 2 бита, которое представляет тип операции
71             ///         throw new NotImplementedException();
72         ///     }
73     /// }
74     /// }
75     ///
76     private struct Transition
77     {
78         ///     public TransitionHeader Header;
79         ///     public Link Source;
80         ///     public Link Linker;
81         ///     public Link Target;
82     /// }
83     ///
84     /// </remarks>
85     public struct Transition : IEquatable<Transition>
86     {
87         public static readonly long Size = Structure<Transition>.Size;
88
89         public readonly ulong TransactionId;
90         public readonly Link<ulong> Before;
91         public readonly Link<ulong> After;
92         public readonly Timestamp Timestamp;
93
94         [MethodImpl(MethodImplOptions.AggressiveInlining)]
95         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
96             ↪ transactionId, Link<ulong> before, Link<ulong> after)
97         {
98             TransactionId = transactionId;
99             Before = before;
100             After = after;
101             Timestamp = uniqueTimestampFactory.Create();
102         }
103
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
106             ↪ transactionId, Link<ulong> before) : this(uniqueTimestampFactory, transactionId,
107             ↪ before, default) { }
108
109         [MethodImpl(MethodImplOptions.AggressiveInlining)]
110         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
111             ↪ transactionId) : this(uniqueTimestampFactory, transactionId, default, default) {
112             ↪ }
113
114         [MethodImpl(MethodImplOptions.AggressiveInlining)]
115         public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
116             ↪ {After}";
117
118         [MethodImpl(MethodImplOptions.AggressiveInlining)]
119         public override bool Equals(object obj) => obj is Transition transition ?
120             ↪ Equals(transition) : false;
121
122         [MethodImpl(MethodImplOptions.AggressiveInlining)]
123         public override int GetHashCode() => (TransactionId, Before, After,
124             ↪ Timestamp).GetHashCode();
125
126         [MethodImpl(MethodImplOptions.AggressiveInlining)]
127         public bool Equals(Transition other) => TransactionId == other.TransactionId &&
128             ↪ Before == other.Before && After == other.After && Timestamp == other.Timestamp;
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>

```

```

129 /// Другие варианты реализации транзакций (атомарности):
130 /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
    → Target)) и индексов.
131 /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
    → потребуется решить вопрос
132 /// со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
    → пересечениями идентификаторов.
133 ///
134 /// Где хранить промежуточный список транзакций?
135 ///
136 /// В оперативной памяти:
137 /// Минусы:
138 /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
139 /// так как нужно отдельно выделять память под список трансформаций.
140 /// 2. Выделенной оперативной памяти может не хватить, в том случае,
141 /// если транзакция использует слишком много трансформаций.
142 /// -> Можно использовать жёсткий диск для слишком длинных транзакций.
143 /// -> Максимальный размер списка трансформаций можно ограничить / задать
    → константой.
144 /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
    → создавая задержку.
145 ///
146 /// На жёстком диске:
147 /// Минусы:
148 /// 1. Длительный отклик, на запись каждой трансформации.
149 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
150 /// -> Это может решаться упаковкой/исключением дублирующих операций.
151 /// -> Также это может решаться тем, что короткие транзакции вообще
152 /// не будут записываться в случае отката.
153 /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
    → операции (трансформации)
    → 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,
206         ↪ Transaction transaction)
207     {
208         layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
209         layer._currentTransactionTransitions = transaction._transitions;
210         layer._currentTransaction = transaction;
211     }
212
213     [MethodImpl(MethodImplOptions.AggressiveInlining)]
214     public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
215     {
216         if (transaction.IsReverted)
217         {
218             throw new InvalidOperationException("Transation is reverted.");
219         }
220         if (transaction.IsCommitted)
221         {
222             throw new InvalidOperationException("Transation is committed.");
223         }
224     }
225
226     [MethodImpl(MethodImplOptions.AggressiveInlining)]
227     protected override void Dispose(bool manual, bool wasDisposed)
228     {
229         if (!wasDisposed && _layer != null && !_layer.Disposable.IsDisposed)
230         {
231             if (!IsCommitted && !IsReverted)
232             {
233                 Revert();
234             }
235             _layer.ResetCurrentTransation();
236         }
237     }
238
239     public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
240
241     private readonly string _logAddress;
242     private readonly FileStream _log;
243     private readonly Queue<Transition> _transitions;
244     private readonly UniqueTimestampFactory _uniqueTimestampFactory;
245     private Task _transitionsPusher;
246     private Transition _lastCommittedTransition;
247     private ulong _currentTransactionId;
248     private Queue<Transition> _currentTransactionTransitions;
249     private Transaction _currentTransaction;
250     private ulong _lastCommittedTransactionId;
251
252     [MethodImpl(MethodImplOptions.AggressiveInlining)]
253     public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
254         : base(links)
255     {
256         if (string.IsNullOrEmpty(logAddress))
257         {
258             throw new ArgumentNullException(nameof(logAddress));
259         }
260         // В первой строке файла хранится последняя закомиченную транзакцию.
261         // При запуске это используется для проверки удачного закрытия файла лога.
262         // In the first line of the file the last committed transaction is stored.
263         // On startup, this is used to check that the log file is successfully closed.
264         var lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
265         var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
266         if (!lastCommittedTransition.Equals(lastWrittenTransition))
267         {
268             Dispose();
269             throw new NotSupportedException("Database is damaged, autorecovery is not
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);

```

```

278         _lastCommittedTransactionId = allTransitions.Length > 0 ? allTransitions.Max(x =>
279             ↪ x.TransactionId) : 0;
280         _uniqueTimestampFactory = new UniqueTimestampFactory();
281         _logAddress = logAddress;
282         _log = FileHelpers.Append(logAddress);
283         _transitions = new Queue<Transition>();
284         _transitionsPusher = new Task(TransitionsPusher);
285         _transitionsPusher.Start();
286     }
287
288     [MethodImpl(MethodImplOptions.AggressiveInlining)]
289     public IList<ulong> GetLinkValue(ulong link) => _links.GetLink(link);
290
291     [MethodImpl(MethodImplOptions.AggressiveInlining)]
292     public override ulong Create(IList<ulong> restrictions)
293     {
294         var createdLinkIndex = _links.Create();
295         var createdLink = new Link<ulong>(_links.GetLink(createdLinkIndex));
296         CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
297             ↪ default, createdLink));
298         return createdLinkIndex;
299     }
300
301     [MethodImpl(MethodImplOptions.AggressiveInlining)]
302     public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
303     {
304         var linkIndex = restrictions[_constants.IndexPart];
305         var beforeLink = new Link<ulong>(_links.GetLink(linkIndex));
306         linkIndex = _links.Update(restrictions, substitution);
307         var afterLink = new Link<ulong>(_links.GetLink(linkIndex));
308         CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
309             ↪ beforeLink, afterLink));
310         return linkIndex;
311     }
312
313     [MethodImpl(MethodImplOptions.AggressiveInlining)]
314     public override void Delete(IList<ulong> restrictions)
315     {
316         var link = restrictions[_constants.IndexPart];
317         var deletedLink = new Link<ulong>(_links.GetLink(link));
318         _links.Delete(link);
319         CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
320             ↪ deletedLink, default));
321     }
322
323     [MethodImpl(MethodImplOptions.AggressiveInlining)]
324     private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
325         ↪ _transitions;
326
327     [MethodImpl(MethodImplOptions.AggressiveInlining)]
328     private void CommitTransition(Transition transition)
329     {
330         if (_currentTransaction != null)
331         {
332             Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
333         }
334         var transitions = GetCurrentTransitions();
335         transitions.Enqueue(transition);
336     }
337
338     [MethodImpl(MethodImplOptions.AggressiveInlining)]
339     private void RevertTransition(Transition transition)
340     {
341         if (transition.After.IsNull()) // Revert Deletion with Creation
342         {
343             _links.Create();
344         }
345         else if (transition.Before.IsNull()) // Revert Creation with Deletion
346         {
347             _links.Delete(transition.After.Index);
348         }
349         else // Revert Update
350         {
351             _links.Update(new[] { transition.After.Index, transition.Before.Source,
352                 ↪ transition.Before.Target });
353         }
354     }

```

```

350 [MethodImpl(MethodImplOptions.AggressiveInlining)]
351 private void ResetCurrentTransaction()
352 {
353     _currentTransactionId = 0;
354     _currentTransactionTransitions = null;
355     _currentTransaction = null;
356 }
357
358 [MethodImpl(MethodImplOptions.AggressiveInlining)]
359 private void PushTransitions()
360 {
361     if (_log == null || _transitions == null)
362     {
363         return;
364     }
365     for (var i = 0; i < _transitions.Count; i++)
366     {
367         var transition = _transitions.Dequeue();
368
369         _log.Write(transition);
370         _lastCommittedTransition = transition;
371     }
372 }
373
374 [MethodImpl(MethodImplOptions.AggressiveInlining)]
375 private void TransitionsPusher()
376 {
377     while (!Disposable.IsDisposed && _transitionsPusher != null)
378     {
379         Thread.Sleep(DefaultPushDelay);
380         PushTransitions();
381     }
382 }
383
384 [MethodImpl(MethodImplOptions.AggressiveInlining)]
385 public Transaction BeginTransaction() => new Transaction(this);
386
387 [MethodImpl(MethodImplOptions.AggressiveInlining)]
388 private void DisposeTransitions()
389 {
390     try
391     {
392         var pusher = _transitionsPusher;
393         if (pusher != null)
394         {
395             _transitionsPusher = null;
396             pusher.Wait();
397         }
398         if (_transitions != null)
399         {
400             PushTransitions();
401         }
402         _log.DisposeIfPossible();
403         FileHelpers.WriteFirst(_logAddress, _lastCommittedTransition);
404     }
405     catch (Exception ex)
406     {
407         ex.Ignore();
408     }
409 }
410
411 #region DisposalBase
412
413 [MethodImpl(MethodImplOptions.AggressiveInlining)]
414 protected override void Dispose(bool manual, bool wasDisposed)
415 {
416     if (!wasDisposed)
417     {
418         DisposeTransitions();
419     }
420     base.Dispose(manual, wasDisposed);
421 }
422
423 #endregion
424 }
425 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Converters;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Unicode
7 {
8     public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
9         ↪ IConverter<char, TLink>
10    {
11        private static readonly UncheckedConverter<char, TLink> _charToAddressConverter =
12            ↪ UncheckedConverter<char, TLink>.Default;
13
14        private readonly IConverter<TLink> _addressToNumberConverter;
15        private readonly TLink _unicodeSymbolMarker;
16
17        [MethodImpl(MethodImplOptions.AggressiveInlining)]
18        public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
19            ↪ addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
20        {
21            _addressToNumberConverter = addressToNumberConverter;
22            _unicodeSymbolMarker = unicodeSymbolMarker;
23        }
24
25        [MethodImpl(MethodImplOptions.AggressiveInlining)]
26        public TLink Convert(char source)
27        {
28            var unaryNumber =
29                ↪ _addressToNumberConverter.Convert(_charToAddressConverter.Convert(source));
30            return _links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
31        }
32    }
33 }

```

1.130 ./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)]

```

```

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

```

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

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Globalization;
4  using System.Runtime.CompilerServices;
5  using System.Text;
6  using Platform.Data.Sequences;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Unicode
11 {
12     public class UnicodeMap
13     {
14         public static readonly ulong FirstCharLink = 1;
15         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
16         public static readonly ulong MapSize = 1 + char.MaxValue;
17
18         private readonly ILinks<ulong> _links;
19         private bool _initialized;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public UnicodeMap(ILinks<ulong> links) => _links = links;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public static UnicodeMap InitNew(ILinks<ulong> links)
26         {

```

```

27     var map = new UnicodeMap(links);
28     map.Init();
29     return map;
30 }
31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 public void Init()
34 {
35     if (!_initialized)
36     {
37         return;
38     }
39     _initialized = true;
40     var firstLink = _links.CreatePoint();
41     if (firstLink != FirstCharLink)
42     {
43         _links.Delete(firstLink);
44     }
45     else
46     {
47         for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
48         {
49             // From NIL to It (NIL -> Character) transformation meaning, (or infinite
50             // ↪ amount of NIL characters before actual Character)
51             var createdLink = _links.CreatePoint();
52             _links.Update(createdLink, firstLink, createdLink);
53             if (createdLink != i)
54             {
55                 throw new InvalidOperationException("Unable to initialize UTF 16
56                 ↪ table.");
57             }
58         }
59     }
60 }
61
62 // 0 - null link
63 // 1 - nil character (0 character)
64 // ...
65 // 65536 (0(1) + 65535 = 65536 possible values)
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 public static ulong FromCharToLink(char character) => (ulong)character + 1;
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 public static char FromLinkToChar(ulong link) => (char)(link - 1);
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 public static bool IsCharLink(ulong link) => link <= MapSize;
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 public static string FromLinksToString(IList<ulong> linksList)
78 {
79     var sb = new StringBuilder();
80     for (int i = 0; i < linksList.Count; i++)
81     {
82         sb.Append(FromLinkToChar(linksList[i]));
83     }
84     return sb.ToString();
85 }
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
89 {
90     var sb = new StringBuilder();
91     if (links.Exists(link))
92     {
93         StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
94             x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
95             ↪ element =>
96             {
97                 sb.Append(FromLinkToChar(element));
98                 return true;
99             });
100     }
101     return sb.ToString();
102 }
103
104 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

102 public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
103     ↪ chars.Length);
104
105 [MethodImpl(MethodImplOptions.AggressiveInlining)]
106 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
107 {
108     // char array to ulong array
109     var linksSequence = new ulong[count];
110     for (var i = 0; i < count; i++)
111     {
112         linksSequence[i] = FromCharToLink(chars[i]);
113     }
114     return linksSequence;
115 }
116
117 [MethodImpl(MethodImplOptions.AggressiveInlining)]
118 public static ulong[] FromStringToLinkArray(string sequence)
119 {
120     // char array to ulong array
121     var linksSequence = new ulong[sequence.Length];
122     for (var i = 0; i < sequence.Length; i++)
123     {
124         linksSequence[i] = FromCharToLink(sequence[i]);
125     }
126     return linksSequence;
127 }
128
129 [MethodImpl(MethodImplOptions.AggressiveInlining)]
130 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
131 {
132     var result = new List<ulong[]>();
133     var offset = 0;
134     while (offset < sequence.Length)
135     {
136         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
137         var relativeLength = 1;
138         var absoluteLength = offset + relativeLength;
139         while (absoluteLength < sequence.Length &&
140             currentCategory ==
141             ↪ CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
142         {
143             relativeLength++;
144             absoluteLength++;
145         }
146         // char array to ulong array
147         var innerSequence = new ulong[relativeLength];
148         var maxLength = offset + relativeLength;
149         for (var i = offset; i < maxLength; i++)
150         {
151             innerSequence[i - offset] = FromCharToLink(sequence[i]);
152         }
153         result.Add(innerSequence);
154         offset += relativeLength;
155     }
156     return result;
157 }
158
159 [MethodImpl(MethodImplOptions.AggressiveInlining)]
160 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
161 {
162     var result = new List<ulong[]>();
163     var offset = 0;
164     while (offset < array.Length)
165     {
166         var relativeLength = 1;
167         if (array[offset] <= LastCharLink)
168         {
169             var currentCategory =
170             ↪ CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
171             var absoluteLength = offset + relativeLength;
172             while (absoluteLength < array.Length &&
173                 array[absoluteLength] <= LastCharLink &&
174                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
175                 ↪ array[absoluteLength])))
176             {
177                 relativeLength++;
178                 absoluteLength++;
179             }
180         }
181     }
182 }

```

```

177         else
178         {
179             var absoluteLength = offset + relativeLength;
180             while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
181             {
182                 relativeLength++;
183                 absoluteLength++;
184             }
185         }
186         // copy array
187         var innerSequence = new ulong[relativeLength];
188         var maxLength = offset + relativeLength;
189         for (var i = offset; i < maxLength; i++)
190         {
191             innerSequence[i - offset] = array[i];
192         }
193         result.Add(innerSequence);
194         offset += relativeLength;
195     }
196     return result;
197 }
198 }
199 }

```

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

```

1  using System;
2  using System.Linq;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5  using Platform.Converters;
6  using Platform.Data.Doublets.Sequences.Walkers;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Unicode
11 {
12     public class UnicodeSequenceToStringConverter<TLink> : LinksOperatorBase<TLink>,
13     ↪ IConverter<TLink, string>
14     {
15         private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
16         private readonly ISequenceWalker<TLink> _sequenceWalker;
17         private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
21         ↪ unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
22         ↪ IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
23         {
24             _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
25             _sequenceWalker = sequenceWalker;
26             _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public string Convert(TLink source)
31         {
32             if (!_unicodeSequenceCriterionMatcher.IsMatched(source))
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 charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.
39             ↪ Convert).ToArray();
40             return new string(charArray);
41         }
42     }
43 }
44 }

```

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

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4  using Platform.Converters;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Unicode
9  {

```



```

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

```

1.135 ./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 }

```

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

```

1 using System;
2 using Xunit;

```

```

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

```

1.137 ./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 }

```

1.138 ./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     }

```

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

```

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

```

```

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;
128
129         links.UseUnicode();
130
131         var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
132
133         var totalSequenceSymbolFrequencyCounter = new
134             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
135
136         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
137             ↪ totalSequenceSymbolFrequencyCounter);
138
139         var index = new
140             ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);

```

```

126     var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
127     ↪
128     var sequenceToItsLocalElementLevelsConverter = new
129     ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
130     ↪ linkToItsFrequencyNumberConverter);
131     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
132     ↪ sequenceToItsLocalElementLevelsConverter);
133
134     var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
135     ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
136
137     ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
138     ↪ index, optimalVariantConverter);
139 }
140
141 private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
142 ↪ SequenceToItsLocalElementLevelsConverter<ulong>
143 ↪ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
144 ↪ OptimalVariantConverter<ulong> optimalVariantConverter)
145 {
146     index.Add(sequence);
147
148     var optimalVariant = optimalVariantConverter.Convert(sequence);
149
150     var readSequence1 = sequences.ToList(optimalVariant);
151
152     Assert.True(sequence.SequenceEqual(readSequence1));
153 }
154
155 [Fact]
156 public static void SavedSequencesOptimizationTest()
157 {
158     LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
159     ↪ (long.MaxValue + 1UL, ulong.MaxValue));
160
161     using (var memory = new HeapResizableDirectMemory())
162     using (var disposableLinks = new UInt64UnitedMemoryLinks(memory,
163     ↪ UInt64UnitedMemoryLinks.DefaultLinksSizeStep, constants, IndexTreeType.Default))
164     {
165         var links = new UInt64Links(disposableLinks);
166
167         var root = links.CreatePoint();
168
169         //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
170         var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
171
172         var unicodeSymbolMarker = links.GetOrCreate(root,
173     ↪ addressToNumberConverter.Convert(1));
174         var unicodeSequenceMarker = links.GetOrCreate(root,
175     ↪ addressToNumberConverter.Convert(2));
176
177         var totalSequenceSymbolFrequencyCounter = new
178     ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
179         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
180     ↪ totalSequenceSymbolFrequencyCounter);
181         var index = new
182     ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
183         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
184         var sequenceToItsLocalElementLevelsConverter = new
185     ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
186     ↪ linkToItsFrequencyNumberConverter);
187         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
188     ↪ sequenceToItsLocalElementLevelsConverter);
189
190         var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
191     ↪ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
192
193         var unicodeSequencesOptions = new SequencesOptions<ulong>()
194     {
195             UseSequenceMarker = true,
196             SequenceMarkerLink = unicodeSequenceMarker,
197             UseIndex = true,
198             Index = index,
199             LinksToSequenceConverter = optimalVariantConverter,
200             Walker = walker,

```



```

45         readSequence2.Add());
46     sw3.Stop();
47
48     Assert.True(sequence.SequenceEqual(readSequence1));
49
50     Assert.True(sequence.SequenceEqual(readSequence2));
51
52     // Assert.True(sw2.Elapsed < sw3.Elapsed);
53
54     Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
55         ↳ {sw2.Elapsed}");
56
57     for (var i = 0; i < sequenceLength; i++)
58     {
59         links.Delete(sequence[i]);
60     }
61 }
62 }
63 }

```

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

```

1  using System.IO;
2  using Xunit;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using Platform.Data.Doublets.Memory.United.Specific;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public static class ResizableDirectMemoryLinksTests
10     {
11         private static readonly LinksConstants<ulong> _constants =
12             ↳ Default<LinksConstants<ulong>>.Instance;
13
14         [Fact]
15         public static void BasicFileMappedMemoryTest()
16         {
17             var tempFilename = Path.GetTempFileName();
18             using (var memoryAdapter = new UInt64UnitedMemoryLinks(tempFilename))
19             {
20                 memoryAdapter.TestBasicMemoryOperations();
21             }
22             File.Delete(tempFilename);
23         }
24
25         [Fact]
26         public static void BasicHeapMemoryTest()
27         {
28             using (var memory = new
29                 ↳ HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
30             using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
31                 ↳ UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
32             {
33                 memoryAdapter.TestBasicMemoryOperations();
34             }
35         }
36
37         private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
38         {
39             var link = memoryAdapter.Create();
40             memoryAdapter.Delete(link);
41         }
42
43         [Fact]
44         public static void NonexistentReferencesHeapMemoryTest()
45         {
46             using (var memory = new
47                 ↳ HeapResizableDirectMemory(UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
48             using (var memoryAdapter = new UInt64UnitedMemoryLinks(memory,
49                 ↳ UInt64UnitedMemoryLinks.DefaultLinksSizeStep))
50             {
51                 memoryAdapter.TestNonexistentReferences();
52             }
53
54             private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
55             {
56                 var link = memoryAdapter.Create();

```

```

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

```

1.142 ./csharp/Platform.Data.Doublets.Tests/ScopeTests.cs

```

1  using Xunit;
2  using Platform.Scopes;
3  using Platform.Memory;
4  using Platform.Data.Doublets.Decorators;
5  using Platform.Reflection;
6  using Platform.Data.Doublets.Memory.United.Generic;
7  using Platform.Data.Doublets.Memory.United.Specific;
8
9  namespace Platform.Data.Doublets.Tests
10 {
11     public static class ScopeTests
12     {
13         [Fact]
14         public static void SingleDependencyTest()
15         {
16             using (var scope = new Scope())
17             {
18                 scope.IncludeAssemblyOf<IMemory>();
19                 var instance = scope.Use<IDirectMemory>();
20                 Assert.IsType<HeapResizableDirectMemory>(instance);
21             }
22         }
23
24         [Fact]
25         public static void CascadeDependencyTest()
26         {
27             using (var scope = new Scope())
28             {
29                 scope.Include<TemporaryFileMappedResizableDirectMemory>();
30                 scope.Include<UInt64UnitedMemoryLinks>();
31                 var instance = scope.Use<ILinks<ulong>>();
32                 Assert.IsType<UInt64UnitedMemoryLinks>(instance);
33             }
34         }
35
36         [Fact]
37         public static void FullAutoResolutionTest()
38         {
39             using (var scope = new Scope(autoInclude: true, autoExplore: true))
40             {
41                 var instance = scope.Use<UInt64Links>();
42                 Assert.IsType<UInt64Links>(instance);
43             }
44         }
45
46         [Fact]
47         public static void TypeParametersTest()
48         {
49             using (var scope = new Scope<Types<HeapResizableDirectMemory,
50                 ↳ UnitedMemoryLinks<ulong>>>())
51             {
52                 var links = scope.Use<ILinks<ulong>>();
53                 Assert.IsType<UnitedMemoryLinks<ulong>>(links);
54             }
55         }
56     }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Linq;
5  using Xunit;

```



```

6 using Platform.Collections;
7 using Platform.Collections.Arrays;
8 using Platform.Random;
9 using Platform.IO;
10 using Platform.Singletons;
11 using Platform.Data.Doublets.Sequences;
12 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
13 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
14 using Platform.Data.Doublets.Sequences.Converters;
15 using Platform.Data.Doublets.Unicode;
16
17 namespace Platform.Data.Doublets.Tests
18 {
19     public static class SequencesTests
20     {
21         private static readonly LinksConstants<ulong> _constants =
22             ↳ Default<LinksConstants<ulong>>.Instance;
23
24         static SequencesTests()
25         {
26             // Trigger static constructor to not mess with performance measurements
27             _ = BitString.GetBitMaskFromIndex(1);
28         }
29
30         [Fact]
31         public static void CreateAllVariantsTest()
32         {
33             const long sequenceLength = 8;
34
35             using (var scope = new TempLinksTestScope(useSequences: true))
36             {
37                 var links = scope.Links;
38                 var sequences = scope.Sequences;
39
40                 var sequence = new ulong[sequenceLength];
41                 for (var i = 0; i < sequenceLength; i++)
42                 {
43                     sequence[i] = links.Create();
44                 }
45
46                 var sw1 = Stopwatch.StartNew();
47                 var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
48
49                 var sw2 = Stopwatch.StartNew();
50                 var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
51
52                 Assert.True(results1.Count > results2.Length);
53                 Assert.True(sw1.Elapsed > sw2.Elapsed);
54
55                 for (var i = 0; i < sequenceLength; i++)
56                 {
57                     links.Delete(sequence[i]);
58                 }
59
60                 Assert.True(links.Count() == 0);
61             }
62
63             //[Fact]
64             //public void CUDTest()
65             //{
66             //    var tempFilename = Path.GetTempFileName();
67
68             //    const long sequenceLength = 8;
69
70             //    const ulong itself = LinksConstants.Itself;
71
72             //    using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
73             //        ↳ DefaultLinksSizeStep))
74             //    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         //         var sequences = new Sequences(links);
85
86         //         var sw1 = Stopwatch.StartNew();
87         //         var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
88
89         //         var sw2 = Stopwatch.StartNew();
90         //         var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
91
92         //         Assert.True(results1.Count > results2.Length);
93         //         Assert.True(sw1.Elapsed > sw2.Elapsed);
94
95         //         for (var i = 0; i < sequenceLength; i++)
96         //             links.Delete(sequence[i]);
97         //     }
98
99     //     File.Delete(tempFilename);
100 //}
101
102 [Fact]
103 public static void AllVariantsSearchTest()
104 {
105     const long sequenceLength = 8;
106
107     using (var scope = new TempLinksTestScope(useSequences: true))
108     {
109         var links = scope.Links;
110         var sequences = scope.Sequences;
111
112         var sequence = new ulong[sequenceLength];
113         for (var i = 0; i < sequenceLength; i++)
114         {
115             sequence[i] = links.Create();
116         }
117
118         var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
119
120         //for (int i = 0; i < createResults.Length; i++)
121         //    sequences.Create(createResults[i]);
122
123         var sw0 = Stopwatch.StartNew();
124         var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
125
126         var sw1 = Stopwatch.StartNew();
127         var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
128
129         var sw2 = Stopwatch.StartNew();
130         var searchResults2 = sequences.Each1(sequence); sw2.Stop();
131
132         var sw3 = Stopwatch.StartNew();
133         var searchResults3 = sequences.Each(sequence.ShiftRight()); sw3.Stop();
134
135         var intersection0 = createResults.Intersect(searchResults0).ToList();
136         Assert.True(intersection0.Count == searchResults0.Count);
137         Assert.True(intersection0.Count == createResults.Length);
138
139         var intersection1 = createResults.Intersect(searchResults1).ToList();
140         Assert.True(intersection1.Count == searchResults1.Count);
141         Assert.True(intersection1.Count == createResults.Length);
142
143         var intersection2 = createResults.Intersect(searchResults2).ToList();
144         Assert.True(intersection2.Count == searchResults2.Count);
145         Assert.True(intersection2.Count == createResults.Length);
146
147         var intersection3 = createResults.Intersect(searchResults3).ToList();
148         Assert.True(intersection3.Count == searchResults3.Count);
149         Assert.True(intersection3.Count == createResults.Length);
150
151         for (var i = 0; i < sequenceLength; i++)
152         {
153             links.Delete(sequence[i]);
154         }
155     }
156 }
157
158 [Fact]
159 public static void BalancedVariantSearchTest()
160 {
161     const long sequenceLength = 200;
162
163     using (var scope = new TempLinksTestScope(useSequences: true))

```

```

164     {
165         var links = scope.Links;
166         var sequences = scope.Sequences;
167
168         var sequence = new ulong[sequenceLength];
169         for (var i = 0; i < sequenceLength; i++)
170         {
171             sequence[i] = links.Create();
172         }
173
174         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
175
176         var sw1 = Stopwatch.StartNew();
177         var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
178
179         var sw2 = Stopwatch.StartNew();
180         var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
181
182         var sw3 = Stopwatch.StartNew();
183         var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
184
185         // На количестве в 200 элементов это будет занимать вечность
186         //var sw4 = Stopwatch.StartNew();
187         //var searchResults4 = sequences.Each(sequence); sw4.Stop();
188
189         Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
190
191         Assert.True(searchResults3.Count == 1 && balancedVariant ==
192             ↪ searchResults3.First());
193
194         //Assert.True(sw1.Elapsed < sw2.Elapsed);
195
196         for (var i = 0; i < sequenceLength; i++)
197         {
198             links.Delete(sequence[i]);
199         }
200     }
201
202     [Fact]
203     public static void AllPartialVariantsSearchTest()
204     {
205         const long sequenceLength = 8;
206
207         using (var scope = new TempLinksTestScope(useSequences: true))
208         {
209             var links = scope.Links;
210             var sequences = scope.Sequences;
211
212             var sequence = new ulong[sequenceLength];
213             for (var i = 0; i < sequenceLength; i++)
214             {
215                 sequence[i] = links.Create();
216             }
217
218             var createResults = sequences.CreateAllVariants2(sequence);
219
220             //var createResultsStrings = createResults.Select(x => x + ": " +
221             ↪ sequences.FormatSequence(x)).ToList();
222             //Global.Trash = createResultsStrings;
223
224             var partialSequence = new ulong[sequenceLength - 2];
225
226             Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
227
228             var sw1 = Stopwatch.StartNew();
229             var searchResults1 =
230             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
231
232             var sw2 = Stopwatch.StartNew();
233             var searchResults2 =
234             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
235
236             //var sw3 = Stopwatch.StartNew();
237             //var searchResults3 =
238             ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
239
240             var sw4 = Stopwatch.StartNew();
241             var searchResults4 =
242             ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();

```

```

238 //Global.Trash = searchResults3;
239
240 //var searchResults1Strings = searchResults1.Select(x => x + ": " +
241 ↪ sequences.FormatSequence(x)).ToList();
242 //Global.Trash = searchResults1Strings;
243
244 var intersection1 = createResults.Intersect(searchResults1).ToList();
245 Assert.True(intersection1.Count == createResults.Length);
246
247 var intersection2 = createResults.Intersect(searchResults2).ToList();
248 Assert.True(intersection2.Count == createResults.Length);
249
250 var intersection4 = createResults.Intersect(searchResults4).ToList();
251 Assert.True(intersection4.Count == createResults.Length);
252
253 for (var i = 0; i < sequenceLength; i++)
254 {
255     links.Delete(sequence[i]);
256 }
257 }
258 }
259
260 [Fact]
261 public static void BalancedPartialVariantsSearchTest()
262 {
263     const long sequenceLength = 200;
264
265     using (var scope = new TempLinksTestScope(useSequences: true))
266     {
267         var links = scope.Links;
268         var sequences = scope.Sequences;
269
270         var sequence = new ulong[sequenceLength];
271         for (var i = 0; i < sequenceLength; i++)
272         {
273             sequence[i] = links.Create();
274         }
275
276         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
277         var balancedVariant = balancedVariantConverter.Convert(sequence);
278
279         var partialSequence = new ulong[sequenceLength - 2];
280
281         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
282
283         var sw1 = Stopwatch.StartNew();
284         var searchResults1 =
285             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
286
287         var sw2 = Stopwatch.StartNew();
288         var searchResults2 =
289             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
290
291         Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
292
293         Assert.True(searchResults2.Count == 1 && balancedVariant ==
294             ↪ searchResults2.First());
295
296         for (var i = 0; i < sequenceLength; i++)
297         {
298             links.Delete(sequence[i]);
299         }
300     }
301
302     [Fact(Skip = "Correct implementation is pending")]
303     public static void PatternMatchTest()
304     {
305         var zeroOrMany = Sequences.Sequences.ZeroOrMany;
306
307         using (var scope = new TempLinksTestScope(useSequences: true))
308         {
309             var links = scope.Links;
310             var sequences = scope.Sequences;
311
312             var e1 = links.Create();
313             var e2 = links.Create();

```

```

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 [Fact]
354 public static void IndexTest()
355 {
356     using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
357         ↪ true }, useSequences: true))
358     {
359         var links = scope.Links;
360         var sequences = scope.Sequences;
361         var index = sequences.Options.Index;
362
363         var e1 = links.Create();
364         var e2 = links.Create();
365
366         var sequence = new[]
367         {
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/Konard/LinksPlatform/%
380 ↪ D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87
381 ↪ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
382 private static readonly string _exampleText =
383     @"([english
384     ↪ version] (https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))

```

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

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

386

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

388

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

390

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

392

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

394

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

396

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

398

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

400

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

402

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

404

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

406

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

408

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

410

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

412

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

414

415 Допустим у нас есть смысл "связать" и смысл "направления", много ли это нам даёт? Много ли
→ вариантов интерпретации? А что если уточнить, каким именно образом выполнена связь? Что если
→ можно задать ей чёткий, конкретный смысл? Что это будет? Тип? Глагол? Связка? Действие?
→ Трансформация? Переход из состояния в состояние? Или всё это и есть объект, суть которого в
→ его конечном состоянии, если конечно конец определён направлением?

```

417  [![белая обычная и направленная связи, чёрная типизированная
    ↳  связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png "белая
    ↳  обычная и направленная связи, чёрная типизированная
    ↳  связь")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png)
418
419  А что если всё это время, мы смотрели на суть как бы снаружи? Можно ли взглянуть на это изнутри?
    ↳  Что будет внутри объектов? Объекты ли это? Или это связи? Может ли эта структура описать
    ↳  сама себя? Но что тогда получится, разве это не рекурсия? Может это фрактал?
420
421  [![белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная
    ↳  связь с рекурсивной внутренней
    ↳  структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png
    ↳  "белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная
    ↳  типизированная связь с рекурсивной внутренней структурой")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png)
422
423  На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
    ↳  рекурсии или фрактала?
424
425  [![белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↳  типизированная связь с двойной рекурсивной внутренней
    ↳  структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png
    ↳  "белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↳  типизированная связь с двойной рекурсивной внутренней структурой")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)
426
427  Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
    ↳  Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
428
429  [![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности,
    ↳  чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://
    ↳  raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png "белая обычная и
    ↳  направленная связи со структурой из 8 цветных элементов последовательности, чёрная
    ↳  типизированная связь со структурой из 8 цветных элементов последовательности")](https://raw
    ↳  .githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)
430
431  ...
432
433  [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animat
    ↳  ion-500.gif
    ↳  "анимация")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
    ↳  -animation-500.gif)";
434
435      private static readonly string _exampleLoremIpsumText =
436          @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
    ↳  incididunt ut labore et dolore magna aliqua.
437  Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
    ↳  consequat.";
438
439      [Fact]
440      public static void CompressionTest()
441      {
442          using (var scope = new TempLinksTestScope(useSequences: true))
443          {
444              var links = scope.Links;
445              var sequences = scope.Sequences;
446
447              var e1 = links.Create();
448              var e2 = links.Create();
449
450              var sequence = new[]
451              {
452                  e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
453              };
454
455              var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
456              var totalSequenceSymbolFrequencyCounter = new
    ↳  TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
457              var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
    ↳  totalSequenceSymbolFrequencyCounter);
458              var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
    ↳  balancedVariantConverter, doubletFrequenciesCache);
459
460              var compressedVariant = compressingConverter.Convert(sequence);
461
462              // 1: [1]          (1->1) point
463              // 2: [2]          (2->2) point
464              // 3: [1,2]        (1->2) doublet
465              // 4: [1,2,1,2]    (3->3) doublet

```

```

466 Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
467 Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
468 Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
469 Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
470
471
472 var source = _constants.SourcePart;
473 var target = _constants.TargetPart;
474
475 Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
476 Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
477 Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
478 Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
479
480 // 4 - length of sequence
481 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
482     ↳ == sequence[0]);
483 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
484     ↳ == sequence[1]);
485 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
486     ↳ == sequence[2]);
487 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
488     ↳ == sequence[3]);
489
490 }
491
492 [Fact]
493 public static void CompressionEfficiencyTest()
494 {
495     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
496         ↳ StringSplitOptions.RemoveEmptyEntries);
497     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
498     var totalCharacters = arrays.Select(x => x.Length).Sum();
499
500     using (var scope1 = new TempLinksTestScope(useSequences: true))
501     using (var scope2 = new TempLinksTestScope(useSequences: true))
502     using (var scope3 = new TempLinksTestScope(useSequences: true))
503     {
504         scope1.Links.Unsync.UseUnicode();
505         scope2.Links.Unsync.UseUnicode();
506         scope3.Links.Unsync.UseUnicode();
507
508         var balancedVariantConverter1 = new
509             ↳ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
510         var totalSequenceSymbolFrequencyCounter = new
511             ↳ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
512         var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
513             ↳ totalSequenceSymbolFrequencyCounter);
514         var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
515             ↳ balancedVariantConverter1, linkFrequenciesCache1,
516             ↳ doInitialFrequenciesIncrement: false);
517
518         //var compressor2 = scope2.Sequences;
519         var compressor3 = scope3.Sequences;
520
521         var constants = Default<LinksConstants<ulong>>.Instance;
522
523         var sequences = compressor3;
524         //var meaningRoot = links.CreatePoint();
525         //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
526         //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
527         //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
528             ↳ constants.Itself);
529
530         //var unaryNumberToAddressConverter = new
531             ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
532         //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
533             ↳ unaryOne);
534         //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
535             ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
536         //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
537             ↳ frequencyPropertyMarker, frequencyMarker);
538         //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
539             ↳ frequencyPropertyOperator, frequencyIncrementer);
540         //var linkToItsFrequencyNumberConverter = new
541             ↳ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
542             ↳ unaryNumberToAddressConverter);

```



```

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

```

```

597         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
598             ↳ scope2.Links.Unsync);
599
600         var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
601             ↳ scope3.Links.Unsync);
602
603         var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
604             ↳ link.IsPartialPoint());
605         var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
606             ↳ link.IsPartialPoint());
607         var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
608             ↳ link.IsPartialPoint());
609
610         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
611             ↳ arrays[i].Length > 3)
612             ↳ Assert.False(structure1 == structure2);
613         //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
614             ↳ arrays[i].Length > 3)
615             ↳ Assert.False(structure3 == structure2);
616
617         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
618         Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
619     }
620
621     Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
622         ↳ totalCharacters);
623     Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
624         ↳ totalCharacters);
625     Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
626         ↳ totalCharacters);
627
628     Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
629         ↳ totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
630         ↳ totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
631         ↳ totalCharacters}");
632
633     Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
634         ↳ scope2.Links.Unsync.Count() - initialCount2);
635     Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
636         ↳ scope2.Links.Unsync.Count() - initialCount2);
637
638     var duplicateProvider1 = new
639         ↳ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
640     var duplicateProvider2 = new
641         ↳ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
642     var duplicateProvider3 = new
643         ↳ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
644
645     var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
646     var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
647     var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
648
649     var duplicates1 = duplicateCounter1.Count();
650
651     ConsoleHelpers.Debug("-----");
652
653     var duplicates2 = duplicateCounter2.Count();
654
655     ConsoleHelpers.Debug("-----");
656
657     var duplicates3 = duplicateCounter3.Count();
658
659     Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
660
661     linkFrequenciesCache1.ValidateFrequencies();
662     linkFrequenciesCache3.ValidateFrequencies();
663 }
664
665 [Fact]
666 public static void CompressionStabilityTest()
667 {
668     // TODO: Fix bug (do a separate test)
669     //const ulong minNumbers = 0;
670     //const ulong maxNumbers = 1000;
671
672     const ulong minNumbers = 10000;

```

```

657     const ulong maxNumbers = 12500;
658
659     var strings = new List<string>();
660
661     for (ulong i = minNumbers; i < maxNumbers; i++)
662     {
663         strings.Add(i.ToString());
664     }
665
666     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
667     var totalCharacters = arrays.Select(x => x.Length).Sum();
668
669     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
        ↳ SequencesOptions<ulong> { UseCompression = true,
        ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
        using (var scope2 = new TempLinksTestScope(useSequences: true))
        {
            scope1.Links.UseUnicode();
            scope2.Links.UseUnicode();
        }
        //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
        var compressor1 = scope1.Sequences;
        var compressor2 = scope2.Sequences;
        var compressed1 = new ulong[arrays.Length];
        var compressed2 = new ulong[arrays.Length];
        var sw1 = Stopwatch.StartNew();
        var START = 0;
        var END = arrays.Length;
        // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
        // Stability issue starts at 10001 or 11000
        //for (int i = START; i < END; i++)
        //{
        //    var first = compressor1.Compress(arrays[i]);
        //    var second = compressor1.Compress(arrays[i]);
        //    if (first == second)
        //        compressed1[i] = first;
        //    else
        //    {
        //        // TODO: Find a solution for this case
        //    }
        //}
        for (int i = START; i < END; i++)
        {
            var first = compressor1.Create(arrays[i].ShiftRight());
            var second = compressor1.Create(arrays[i].ShiftRight());
            if (first == second)
            {
                compressed1[i] = first;
            }
            else
            {
                // TODO: Find a solution for this case
            }
        }
        var elapsed1 = sw1.Elapsed;
        var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
        var sw2 = Stopwatch.StartNew();
        for (int i = START; i < END; i++)
        {
            var first = balancedVariantConverter.Convert(arrays[i]);
            var second = balancedVariantConverter.Convert(arrays[i]);
            if (first == second)
            {
                compressed2[i] = first;
            }
        }
        var elapsed2 = sw2.Elapsed;

```

```

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

```

```

802     scope2.Links.UseUnicode();
803
804     var compressor1 = scope1.Sequences;
805     var compressor2 = scope2.Sequences;
806
807     var compressed1 = new ulong[arrays.Length];
808     var compressed2 = new ulong[arrays.Length];
809
810     var sw1 = Stopwatch.StartNew();
811
812     var START = 0;
813     var END = arrays.Length;
814
815     for (int i = START; i < END; i++)
816     {
817         compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
818     }
819
820     var elapsed1 = sw1.Elapsed;
821
822     var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
823
824     var sw2 = Stopwatch.StartNew();
825
826     for (int i = START; i < END; i++)
827     {
828         compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
829     }
830
831     var elapsed2 = sw2.Elapsed;
832
833     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
834         ↳ {elapsed2}");
835
836     Assert.True(elapsed1 > elapsed2);
837
838     // Checks
839     for (int i = START; i < END; i++)
840     {
841         var sequence1 = compressed1[i];
842         var sequence2 = compressed2[i];
843
844         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
845         {
846             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
847                 ↳ scope1.Links);
848
849             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
850                 ↳ scope2.Links);
851
852             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
853         }
854     }
855
856     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
857     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
858
859     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
860         ↳ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
861         ↳ totalCharacters}}");
862
863     // Can be worse than balanced variant
864     //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
865
866     //compressor1.ValidateFrequencies();
867 }
868
869 [Fact]
870 public static void AllTreeBreakDownAtSequencesCreationBugTest()
871 {
872     // Made out of AllPossibleConnectionsTest test.
873
874     //const long sequenceLength = 5; //100% bug
875     const long sequenceLength = 4; //100% bug
876     //const long sequenceLength = 3; //100% _no_bug (ok)
877
878     using (var scope = new TempLinksTestScope(useSequences: true))
879     {
880         var links = scope.Links;

```

```

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

```

```

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

```

1.144 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs

```

1  using System;
2  using Xunit;
3  using Platform.Memory;
4  using Platform.Data.Doublets.Memory.Split.Generic;
5
6  namespace Platform.Data.Doublets.Tests
7  {
8      public unsafe static class SplitMemoryGenericLinksTests
9      {
10         [Fact]
11         public static void CRUDTest()
12         {
13             Using<byte>(links => links.TestCRUDOperations());
14             Using<ushort>(links => links.TestCRUDOperations());
15             Using<uint>(links => links.TestCRUDOperations());
16             Using<ulong>(links => links.TestCRUDOperations());
17         }
18
19         [Fact]
20         public static void RawNumbersCRUDTest()
21         {
22             UsingWithExternalReferences<byte>(links => links.TestRawNumbersCRUDOperations());
23             UsingWithExternalReferences<ushort>(links => links.TestRawNumbersCRUDOperations());
24             UsingWithExternalReferences<uint>(links => links.TestRawNumbersCRUDOperations());
25             UsingWithExternalReferences<ulong>(links => links.TestRawNumbersCRUDOperations());
26         }
27
28         [Fact]
29         public static void MultipleRandomCreationsAndDeletionsTest()
30         {
31             Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
32             ↪ MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
33             ↪ implementation of tree cuts out 5 bits from the address space.
34             Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
35             ↪ stMultipleRandomCreationsAndDeletions(100));
36             Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
37             ↪ MultipleRandomCreationsAndDeletions(100));

```

```

34         Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMulti
        ↪ tMultipleRandomCreationsAndDeletions(100));
35     }
36
37     private static void Using<TLink>(Action<ILinks<TLink>> action)
38     {
39         using (var dataMemory = new HeapResizableDirectMemory())
40         using (var indexMemory = new HeapResizableDirectMemory())
41         using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory))
42         {
43             action(memory);
44         }
45     }
46
47     private static void UsingWithExternalReferences<TLink>(Action<ILinks<TLink>> action)
48     {
49         var constants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
50         using (var dataMemory = new HeapResizableDirectMemory())
51         using (var indexMemory = new HeapResizableDirectMemory())
52         using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory,
        ↪ SplitMemoryLinks<TLink>.DefaultLinksSizeStep, constants))
53         {
54             action(memory);
55         }
56     }
57 }
58 }

```

1.145 ./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().TestMultipl
        ↪ eRandomCreationsAndDeletions(100));
27         }
28
29         private static void Using(Action<ILinks<TLink>> action)
30         {
31             using (var dataMemory = new HeapResizableDirectMemory())
32             using (var indexMemory = new HeapResizableDirectMemory())
33             using (var memory = new UInt32SplitMemoryLinks(dataMemory, indexMemory))
34             {
35                 action(memory);
36             }
37         }
38
39         private static void UsingWithExternalReferences(Action<ILinks<TLink>> action)
40         {
41             var constants = new LinksConstants<TLink>(enableExternalReferencesSupport: true);
42             using (var dataMemory = new HeapResizableDirectMemory())
43             using (var indexMemory = new HeapResizableDirectMemory())
44             using (var memory = new UInt32SplitMemoryLinks(dataMemory, indexMemory,
        ↪ UInt32SplitMemoryLinks.DefaultLinksSizeStep, constants))
45             {
46                 action(memory);
47             }
48         }
49     }
50 }

```



```

48     }
49 }
50 }

```

1.146 ./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt64LinksTests.cs

```

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

```

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

```

1  using System.IO;
2  using Platform.Disposables;
3  using Platform.Data.Doublets.Sequences;
4  using Platform.Data.Doublets.Decorators;
5  using Platform.Data.Doublets.Memory.United.Specific;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public class TempLinksTestScope : DisposableBase
10     {
11         public ILinks<ulong> MemoryAdapter { get; }
12         public SynchronizedLinks<ulong> Links { get; }
13         public Sequences.Sequences Sequences { get; }
14         public string TempFilename { get; }
15         public string TempTransactionLogFilename { get; }
16         private readonly bool _deleteFiles;
17
18         public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
19             ↪ useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
20             ↪ useLog) { }

```

```

19
20 public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
    ↳ true, bool useSequences = false, bool useLog = false)
21 {
22     _deleteFiles = deleteFiles;
23     TempFilename = Path.GetTempFileName();
24     TempTransactionLogFilename = Path.GetTempFileName();
25     var coreMemoryAdapter = new UInt64UnitedMemoryLinks(TempFilename);
26     MemoryAdapter = useLog ? (ILinks<ulong>)new
        ↳ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
        ↳ coreMemoryAdapter;
27     Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
28     if (useSequences)
29     {
30         Sequences = new Sequences.Sequences(Links, sequencesOptions);
31     }
32 }
33
34 protected override void Dispose(bool manual, bool wasDisposed)
35 {
36     if (!wasDisposed)
37     {
38         Links.Unsync.DisposeIfPossible();
39         if (_deleteFiles)
40         {
41             DeleteFiles();
42         }
43     }
44 }
45
46 public void DeleteFiles()
47 {
48     File.Delete(TempFilename);
49     File.Delete(TempTransactionLogFilename);
50 }
51 }
52 }

```

1.148 ./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs

```

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

```

```

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

```

```

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

```

```

196         }
197     }
198     Assert.True(addressToUInt64Converter.Convert(links.Count()) == 0L);
199 }
200 }
201 }
202 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.IO;
5  using System.Text;
6  using System.Threading;
7  using System.Threading.Tasks;
8  using Xunit;
9  using Platform.Disposables;
10 using Platform.Ranges;
11 using Platform.Random;
12 using Platform.Timestamps;
13 using Platform.Reflection;
14 using Platform.Singletons;
15 using Platform.Scopes;
16 using Platform.Counters;
17 using Platform.Diagnostics;
18 using Platform.IO;
19 using Platform.Memory;
20 using Platform.Data.Doublets.Decorators;
21 using Platform.Data.Doublets.Memory.United.Specific;
22
23 namespace Platform.Data.Doublets.Tests
24 {
25     public static class UInt64LinksTests
26     {
27         private static readonly LinksConstants<ulong> _constants =
28             ↪ Default<LinksConstants<ulong>>.Instance;
29
30         private const long Iterations = 10 * 1024;
31
32         #region Concept
33
34         [Fact]
35         public static void MultipleCreateAndDeleteTest()
36         {
37             using (var scope = new Scope<Types<HeapResizableDirectMemory,
38                 ↪ UInt64UnitedMemoryLinks>>())
39             {
40                 new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeletions(100);
41             }
42         }
43
44         [Fact]
45         public static void CascadeUpdateTest()
46         {
47             var itself = _constants.Itself;
48             using (var scope = new TempLinksTestScope(useLog: true))
49             {
50                 var links = scope.Links;
51
52                 var l1 = links.Create();
53                 var l2 = links.Create();
54
55                 l2 = links.Update(l2, l2, l1, l2);
56
57                 links.CreateAndUpdate(l2, itself);
58                 links.CreateAndUpdate(l2, itself);
59
60                 l2 = links.Update(l2, l1);
61
62                 links.Delete(l2);
63
64                 Global.Trash = links.Count();
65
66                 links.Unsync.DisposeIfPossible(); // Close links to access log
67
68                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
69             }
70         }
71     }
72 }

```

```

69 [Fact]
70 public static void BasicTransactionLogTest()
71 {
72     using (var scope = new TempLinksTestScope(useLog: true))
73     {
74         var links = scope.Links;
75         var l1 = links.Create();
76         var l2 = links.Create();
77
78         Global.Trash = links.Update(l2, l2, l1, l2);
79
80         links.Delete(l1);
81
82         links.Unsync.DisposeIfPossible(); // Close links to access log
83
84         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
85             ↪ e.TempTransactionLogFilename);
86     }
87 }
88
89 [Fact]
90 public static void TransactionAutoRevertedTest()
91 {
92     // Auto Reverted (Because no commit at transaction)
93     using (var scope = new TempLinksTestScope(useLog: true))
94     {
95         var links = scope.Links;
96         var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
97         using (var transaction = transactionsLayer.BeginTransaction())
98         {
99             var l1 = links.Create();
100             var l2 = links.Create();
101
102             links.Update(l2, l2, l1, l2);
103         }
104
105         Assert.Equal(0UL, links.Count());
106
107         links.Unsync.DisposeIfPossible();
108
109         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
110             ↪ cope.TempTransactionLogFilename);
111         Assert.Single(transitions);
112     }
113 }
114
115 [Fact]
116 public static void TransactionUserCodeErrorNoDataSavedTest()
117 {
118     // User Code Error (Autoreverted), no data saved
119     var itself = _constants.Itself;
120
121     TempLinksTestScope lastScope = null;
122     try
123     {
124         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
125             ↪ useLog: true))
126         {
127             var links = scope.Links;
128             var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
129                 ↪ atorBase<ulong>)links.Unsync).Links;
130             using (var transaction = transactionsLayer.BeginTransaction())
131             {
132                 var l1 = links.CreateAndUpdate(itself, itself);
133                 var l2 = links.CreateAndUpdate(itself, itself);
134
135                 l2 = links.Update(l2, l2, l1, l2);
136
137                 links.CreateAndUpdate(l2, itself);
138                 links.CreateAndUpdate(l2, itself);
139
140                 //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
141                 ↪ tion>(scope.TempTransactionLogFilename);
142
143                 l2 = links.Update(l2, l1);
144
145                 links.Delete(l2);

```

```

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

```

```

218 [Fact]
219 public static void TransactionCommit()
220 {
221     var itself = _constants.Itself;
222
223     var tempDatabaseFilename = Path.GetTempFileName();
224     var tempTransactionLogFilename = Path.GetTempFileName();
225
226     // Commit
227     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
228         ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
229     using (var links = new UInt64Links(memoryAdapter))
230     {
231         using (var transaction = memoryAdapter.BeginTransaction())
232         {
233             var l1 = links.CreateAndUpdate(itself, itself);
234             var l2 = links.CreateAndUpdate(itself, itself);
235
236             Global.Trash = links.Update(l2, l2, l1, l2);
237
238             links.Delete(l1);
239
240             transaction.Commit();
241         }
242
243         Global.Trash = links.Count();
244     }
245
246     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
247         ↪ sactionLogFilename);
248 }
249 [Fact]
250 public static void TransactionDamage()
251 {
252     var itself = _constants.Itself;
253
254     var tempDatabaseFilename = Path.GetTempFileName();
255     var tempTransactionLogFilename = Path.GetTempFileName();
256
257     // Commit
258     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
259         ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
260     using (var links = new UInt64Links(memoryAdapter))
261     {
262         using (var transaction = memoryAdapter.BeginTransaction())
263         {
264             var l1 = links.CreateAndUpdate(itself, itself);
265             var l2 = links.CreateAndUpdate(itself, itself);
266
267             Global.Trash = links.Update(l2, l2, l1, l2);
268
269             links.Delete(l1);
270
271             transaction.Commit();
272         }
273
274         Global.Trash = links.Count();
275     }
276
277     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
278         ↪ sactionLogFilename);
279
280     // Damage database
281     FileHelpers.WriteFirst(tempTransactionLogFilename, new
282         ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
283
284     // Try load damaged database
285     try
286     {
287         // TODO: Fix
288         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
289             ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
290         using (var links = new UInt64Links(memoryAdapter))
291         {
292             Global.Trash = links.Count();
293         }
294     }

```



```

291     }
292     catch (NotSupportedException ex)
293     {
294         Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
        ↳ yet.");
295     }
296
297     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
        ↳ sactionLogFilename);
298
299     File.Delete(tempDatabaseFilename);
300     File.Delete(tempTransactionLogFilename);
301 }
302
303 [Fact]
304 public static void Bug1Test()
305 {
306     var tempDatabaseFilename = Path.GetTempFileName();
307     var tempTransactionLogFilename = Path.GetTempFileName();
308
309     var itself = _constants.Itself;
310
311     // User Code Error (Autoreverted), some data saved
312     try
313     {
314         ulong l1;
315         ulong l2;
316
317         using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
318         using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
        ↳ tempTransactionLogFilename))
319         using (var links = new UInt64Links(memoryAdapter))
320         {
321             l1 = links.CreateAndUpdate(itself, itself);
322             l2 = links.CreateAndUpdate(itself, itself);
323
324             l2 = links.Update(l2, l2, l1, l2);
325
326             links.CreateAndUpdate(l2, itself);
327             links.CreateAndUpdate(l2, itself);
328         }
329
330         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp_
        ↳ TransactionLogFilename);
331
332         using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
333         using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
        ↳ tempTransactionLogFilename))
334         using (var links = new UInt64Links(memoryAdapter))
335         {
336             using (var transaction = memoryAdapter.BeginTransaction())
337             {
338                 l2 = links.Update(l2, l1);
339
340                 links.Delete(l2);
341
342                 ExceptionThrower();
343
344                 transaction.Commit();
345             }
346
347             Global.Trash = links.Count();
348         }
349     }
350     catch
351     {
352         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp_
        ↳ TransactionLogFilename);
353     }
354
355     File.Delete(tempDatabaseFilename);
356     File.Delete(tempTransactionLogFilename);
357 }
358
359 private static void ExceptionThrower() => throw new InvalidOperationException();
360
361 [Fact]
362 public static void PathsTest()
363 {

```

```

364     var source = _constants.SourcePart;
365     var target = _constants.TargetPart;
366
367     using (var scope = new TempLinksTestScope())
368     {
369         var links = scope.Links;
370         var l1 = links.CreatePoint();
371         var l2 = links.CreatePoint();
372
373         var r1 = links.GetByKeys(l1, source, target, source);
374         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
375     }
376 }
377
378 [Fact]
379 public static void RecursiveStringFormattingTest()
380 {
381     using (var scope = new TempLinksTestScope(useSequences: true))
382     {
383         var links = scope.Links;
384         var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
385
386         var a = links.CreatePoint();
387         var b = links.CreatePoint();
388         var c = links.CreatePoint();
389
390         var ab = links.GetOrCreate(a, b);
391         var cb = links.GetOrCreate(c, b);
392         var ac = links.GetOrCreate(a, c);
393
394         a = links.Update(a, c, b);
395         b = links.Update(b, a, c);
396         c = links.Update(c, a, b);
397
398         Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
399         Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
400         Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
401
402         Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
403             ↪ "(5:(4:5 (6:5 4)) 6)");
404         Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
405             ↪ "(6:(5:(4:5 6) 6) 4)");
406         Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
407             ↪ "(4:(5:4 (6:5 4)) 6)");
408
409         // TODO: Think how to build balanced syntax tree while formatting structure (eg.
410         ↪ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)"
411
412         Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
413             ↪ "{{5}{5}{4}{6}}");
414         Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
415             ↪ "{{5}{6}{6}{4}}");
416         Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
417             ↪ "{{4}{5}{4}{6}}");
418     }
419 }
420
421 private static void DefaultFormatter(StringBuilder sb, ulong link)
422 {
423     sb.Append(link.ToString());
424 }
425
426 #endregion
427
428 #region Performance
429
430 /*
431 public static void RunAllPerformanceTests()
432 {
433     try
434     {
435         links.TestLinksInSteps();
436     }
437     catch (Exception ex)
438     {
439         ex.WriteToConsole();
440     }
441 }
442
443 return;

```

```

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

```

```

513         for (int i = 0; i < loops; i++)
514         {
515             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
516             ↪ 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,
524             ↪ searchMeasurements[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
525         }
526
527         ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
528         ↪ links.Total);
529     }
530
531     private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
532     ↪ amountToCreate)
533     {
534         for (long i = 0; i < amountToCreate; i++)
535             links.Create(0, 0);
536     }
537
538     private static TimeSpan GetBaseRandomLoopOverhead(long loops)
539     {
540         return Measure(() =>
541         {
542             ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
543             ulong result = 0;
544             for (long i = 0; i < loops; i++)
545             {
546                 var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
547                 var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
548                 result += maxValue + source + target;
549             }
550             Global.Trash = result;
551         });
552     }
553
554     /*
555     [Fact(Skip = "performance test")]
556     public static void GetSourceTest()
557     {
558         using (var scope = new TempLinksTestScope())
559         {
560             var links = scope.Links;
561             ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
562             ↪ Iterations);
563
564             ulong counter = 0;
565
566             //var firstLink = links.First();
567             // Создаём одну связь, из которой будет производить считывание
568             var firstLink = links.Create();
569
570             var sw = Stopwatch.StartNew();
571
572             // Тестируем саму функцию
573             for (ulong i = 0; i < Iterations; i++)
574             {
575                 counter += links.GetSource(firstLink);
576             }
577
578             var elapsedTime = sw.Elapsed;
579
580             var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
581
582             // Удаляем связь, из которой производилось считывание
583             links.Delete(firstLink);
584
585             ConsoleHelpers.Debug(
586                 "{0} Iterations of GetSource function done in {1} ({2} Iterations per
587                 ↪ second), counter result: {3}",
588                 Iterations, elapsedTime, (long)iterationsPerSecond, counter);
589         }
590     }

```

```
[Fact(Skip = "performance test")]
public static void GetSourceInParallel()
{
    using (var scope = new TempLinksTestScope())
    {
        var links = scope.Links;
        ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
        ↪ parallel.", Iterations);

        long counter = 0;

        //var firstLink = links.First();
        var firstLink = links.Create();

        var sw = Stopwatch.StartNew();

        // Тестируем саму функцию
        Parallel.For(0, Iterations, x =>
        {
            Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
            //Interlocked.Increment(ref counter);
        });

        var elapsedTime = sw.Elapsed;

        var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

        links.Delete(firstLink);

        ConsoleHelpers.Debug(
            "{0} Iterations of GetSource function done in {1} ({2} Iterations per
            ↪ second), counter result: {3}",
            Iterations, elapsedTime, (long)iterationsPerSecond, counter);
    }
}

[Fact(Skip = "performance test")]
public static void TestGetTarget()
{
    using (var scope = new TempLinksTestScope())
    {
        var links = scope.Links;
        ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
        ↪ Iterations);

        ulong counter = 0;

        //var firstLink = links.First();
        var firstLink = links.Create();

        var sw = Stopwatch.StartNew();

        for (ulong i = 0; i < Iterations; i++)
        {
            counter += links.GetTarget(firstLink);
        }

        var elapsedTime = sw.Elapsed;

        var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

        links.Delete(firstLink);

        ConsoleHelpers.Debug(
            "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
            ↪ second), counter result: {3}",
            Iterations, elapsedTime, (long)iterationsPerSecond, counter);
    }
}

[Fact(Skip = "performance test")]
public static void TestGetTargetInParallel()
{
    using (var scope = new TempLinksTestScope())
    {
        var links = scope.Links;
        ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
        ↪ parallel.", Iterations);
```

```

661         long counter = 0;
662
663
664         //var firstLink = links.First();
665         var firstLink = links.Create();
666
667         var sw = Stopwatch.StartNew();
668
669         Parallel.For(0, Iterations, x =>
670         {
671             Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
672             //Interlocked.Increment(ref counter);
673         });
674
675         var elapsedTime = sw.Elapsed;
676
677         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
678
679         links.Delete(firstLink);
680
681         ConsoleHelpers.Debug(
682             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↪ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
683     }
684 }
685
686
687 // TODO: Заполнить базу данных перед тестом
688 /*
689 [Fact]
690 public void TestRandomSearchFixed()
691 {
692     var tempFilename = Path.GetTempFileName();
693
694     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
        ↪ DefaultLinksSizeStep))
695     {
696         long iterations = 64 * 1024 * 1024 /
        ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
697
698         ulong counter = 0;
699         var maxLink = links.Total;
700
701         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
702
703         var sw = Stopwatch.StartNew();
704
705         for (var i = iterations; i > 0; i--)
706         {
707             var source =
        ↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
708             var target =
        ↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
709
710             counter += links.Search(source, target);
711         }
712
713         var elapsedTime = sw.Elapsed;
714
715         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
716
717         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
        ↪ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
        ↪ counter);
718     }
719
720     File.Delete(tempFilename);
721 }*/
722
723 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
724 public static void TestRandomSearchAll()
725 {
726     using (var scope = new TempLinksTestScope())
727     {
728         var links = scope.Links;
729         ulong counter = 0;
730
731         var maxLink = links.Count();
732
733         var iterations = links.Count();

```

```

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

```

```

808         File.Delete(tempFilename);
809     }
810     */
811
812     /*
813     [Fact]
814     public static void TestParallelForeach()
815     {
816         var tempFilename = Path.GetTempFileName();
817
818         using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↵ DefaultLinksSizeStep))
819         {
820
821             long counter = 0;
822
823             ConsoleHelpers.Debug("Testing parallel foreach through links.");
824
825             var sw = Stopwatch.StartNew();
826
827             //Parallel.ForEach((IEnumerable<ulong>)links, x =>
828             //{
829             //    Interlocked.Increment(ref counter);
830             //});
831
832             var elapsedTime = sw.Elapsed;
833
834             var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
835
836             ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↵ {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
837         }
838
839         File.Delete(tempFilename);
840     }
841     */
842
843     [Fact(Skip = "performance test")]
844     public static void Create64BillionLinks()
845     {
846         using (var scope = new TempLinksTestScope())
847         {
848             var links = scope.Links;
849             var linksBeforeTest = links.Count();
850
851             long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;
852
853             ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
854
855             var elapsedTime = Performance.Measure(() =>
856             {
857                 for (long i = 0; i < linksToCreate; i++)
858                 {
859                     links.Create();
860                 }
861             });
862
863             var linksCreated = links.Count() - linksBeforeTest;
864             var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
865
866             ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
867
868             ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
↵ linksCreated, elapsedTime,
869                 (long)linksPerSecond);
870         }
871     }
872
873     [Fact(Skip = "performance test")]
874     public static void Create64BillionLinksInParallel()
875     {
876         using (var scope = new TempLinksTestScope())
877         {
878             var links = scope.Links;
879             var linksBeforeTest = links.Count();
880
881             var sw = Stopwatch.StartNew();
882
883             long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;
884

```



```

885     ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
886
887     Parallel.For(0, linksToCreate, x => links.Create());
888
889     var elapsedTime = sw.Elapsed;
890
891     var linksCreated = links.Count() - linksBeforeTest;
892     var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
893
894     ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
895         ↪ linksCreated, elapsedTime,
896         ↪ (long)linksPerSecond);
897 }
898
899 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
900 public static void TestDeletionOfAllLinks()
901 {
902     using (var scope = new TempLinksTestScope())
903     {
904         var links = scope.Links;
905         var linksBeforeTest = links.Count();
906
907         ConsoleHelpers.Debug("Deleting all links");
908
909         var elapsedTime = Performance.Measure(links.DeleteAll);
910
911         var linksDeleted = linksBeforeTest - links.Count();
912         var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
913
914         ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
915             ↪ linksDeleted, elapsedTime,
916             ↪ (long)linksPerSecond);
917     }
918 }
919 #endregion
920 }
921 }

```

1.150 ./csharp/Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs

```

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

```

```

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

```

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

```

1  using Xunit;
2  using Platform.Converters;
3  using Platform.Memory;
4  using Platform.Reflection;
5  using Platform.Scopes;
6  using Platform.Data.Numbers.Raw;
7  using Platform.Data.Doublets.Incrementers;
8  using Platform.Data.Doublets.Numbers.Unary;
9  using Platform.Data.Doublets.PropertyOperators;
10 using Platform.Data.Doublets.Sequences.Converters;
11 using Platform.Data.Doublets.Sequences.Indexes;
12 using Platform.Data.Doublets.Sequences.Walkers;
13 using Platform.Data.Doublets.Unicode;
14 using Platform.Data.Doublets.Memory.United.Generic;
15 using Platform.Data.Doublets.CriterionMatchers;
16
17 namespace Platform.Data.Doublets.Tests
18 {
19     public static class UnicodeConvertersTests
20     {
21         [Fact]
22         public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
23         {
24             using (var scope = new TempLinksTestScope())
25             {
26                 var links = scope.Links;
27                 var meaningRoot = links.CreatePoint();
28                 var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
29                 var powerOf2ToUnaryNumberConverter = new
    ↪     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);

```

```
}
```

```
[Fact]
```

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

```
{
```

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

```
    {
```

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

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

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

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

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

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

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

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

```
        var powerOf2ToUnaryNumberConverter = new
```

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

```
        var addressToUnaryNumberConverter = new
```

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

```
        var charToUnicodeSymbolConverter = new
```

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

```
        ↪ unicodeSymbolMarker);
```

```
        var unaryNumberToAddressConverter = new
```

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

```
        ↪ powerOf2ToUnaryNumberConverter);
```

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

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

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

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

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

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

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

```
        var linkToItsFrequencyNumberConverter = new
```

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

```
        ↪ unaryNumberToAddressConverter);
```

```
        var sequenceToItsLocalElementLevelsConverter = new
```

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

```
        ↪ linkToItsFrequencyNumberConverter);
```

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

```
        ↪ sequenceToItsLocalElementLevelsConverter);
```

```
        var stringToUnicodeSequenceConverter = new
```

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

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

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

```
        var unicodeSequenceLink =
```

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

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

```
        ↪ unicodeSymbolMarker);
```

```
        var unicodeSymbolToCharConverter = new
```

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

```
        ↪ unicodeSymbolCriterionMatcher);
```

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

```
        ↪ unicodeSequenceMarker);
```

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

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

```
        var unicodeSequenceToStringConverter = new
```

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

```
        ↪ unicodeSequenceCriterionMatcher, sequenceWalker,
```

```
        ↪ unicodeSymbolToCharConverter);
```

```
        var resultingString =
```

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

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

```
    }
```

```
}
```

```
}
```

```
111 }
```

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

```
1 using System;
2 using Xunit;
3 using Platform.Reflection;
4 using Platform.Memory;
5 using Platform.Scopes;
6 using Platform.Data.Doublets.Memory.United.Specific;
7 using TLink = System.UInt32;
8
9 namespace Platform.Data.Doublets.Tests
10 {
11     public unsafe static class UnitedMemoryUInt32LinksTests
12     {
13         [Fact]
14         public static void CRUDTest()
15         {
16             Using(links => links.TestCRUDOperations());
17         }
18
19         [Fact]
20         public static void RawNumbersCRUDTest()
21         {
22             Using(links => links.TestRawNumbersCRUDOperations());
23         }
24
25         [Fact]
26         public static void MultipleRandomCreationsAndDeletionsTest()
27         {
28             Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(100));
29         }
30
31         private static void Using(Action<ILinks<TLink>> action)
32         {
33             using (var scope = new Scope<Types<HeapResizableDirectMemory,
34                 ↳ UInt32UnitedMemoryLinks>>())
35             {
36                 action(scope.Use<ILinks<TLink>>());
37             }
38         }
39     }
40 }
```

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

```
1 using System;
2 using Xunit;
3 using Platform.Reflection;
4 using Platform.Memory;
5 using Platform.Scopes;
6 using Platform.Data.Doublets.Memory.United.Specific;
7 using TLink = System.UInt64;
8
9 namespace Platform.Data.Doublets.Tests
10 {
11     public unsafe static class UnitedMemoryUInt64LinksTests
12     {
13         [Fact]
14         public static void CRUDTest()
15         {
16             Using(links => links.TestCRUDOperations());
17         }
18
19         [Fact]
20         public static void RawNumbersCRUDTest()
21         {
22             Using(links => links.TestRawNumbersCRUDOperations());
23         }
24
25         [Fact]
26         public static void MultipleRandomCreationsAndDeletionsTest()
27         {
28             Using(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().TestMultipleRandomCreationsAndDeletions(100));
29         }
30
31         private static void Using(Action<ILinks<TLink>> action)
32         {
33             using (var scope = new Scope<Types<HeapResizableDirectMemory,
34                 ↳ UInt64UnitedMemoryLinks>>())
35             {
36                 action(scope.Use<ILinks<TLink>>());
37             }
38         }
39     }
40 }
```

```
33         using (var scope = new Scope<Types<HeapResizableDirectMemory,  
34             ↪ UInt64UnitedMemoryLinks>>())  
35         {  
36             action(scope.Use<ILinks<TLink>>());  
37         }  
38     }  
39 }
```

Index

`./csharp/Platform.Data.Doublets.Tests/GenericLinksTests.cs`, 193
`./csharp/Platform.Data.Doublets.Tests/ILinksExtensionsTests.cs`, 194
`./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs`, 194
`./csharp/Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs`, 195
`./csharp/Platform.Data.Doublets.Tests/ReadSequenceTests.cs`, 198
`./csharp/Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs`, 199
`./csharp/Platform.Data.Doublets.Tests/ScopeTests.cs`, 200
`./csharp/Platform.Data.Doublets.Tests/SequencesTests.cs`, 200
`./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs`, 215
`./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt32LinksTests.cs`, 216
`./csharp/Platform.Data.Doublets.Tests/SplitMemoryUInt64LinksTests.cs`, 217
`./csharp/Platform.Data.Doublets.Tests/TempLinksTestScope.cs`, 217
`./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs`, 218
`./csharp/Platform.Data.Doublets.Tests/UInt64LinksTests.cs`, 221
`./csharp/Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs`, 233
`./csharp/Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs`, 234
`./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt32LinksTests.cs`, 236
`./csharp/Platform.Data.Doublets.Tests/UnitedMemoryUInt64LinksTests.cs`, 236
`./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/UInt64Links.cs`, 6
`./csharp/Platform.Data.Doublets/Decorators/UniLinks.cs`, 7
`./csharp/Platform.Data.Doublets/Doublet.cs`, 12
`./csharp/Platform.Data.Doublets/DoubletComparer.cs`, 13
`./csharp/Platform.Data.Doublets/ILinks.cs`, 13
`./csharp/Platform.Data.Doublets/ILinksExtensions.cs`, 13
`./csharp/Platform.Data.Doublets/ISynchronizedLinks.cs`, 25
`./csharp/Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs`, 25
`./csharp/Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs`, 26
`./csharp/Platform.Data.Doublets/Link.cs`, 26
`./csharp/Platform.Data.Doublets/LinkExtensions.cs`, 30
`./csharp/Platform.Data.Doublets/LinksOperatorBase.cs`, 30
`./csharp/Platform.Data.Doublets/Memory/ILinksListMethods.cs`, 30
`./csharp/Platform.Data.Doublets/Memory/ILinksTreeMethods.cs`, 30
`./csharp/Platform.Data.Doublets/Memory/IndexTreeType.cs`, 31
`./csharp/Platform.Data.Doublets/Memory/LinksHeader.cs`, 31
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSizeBalancedTreeMethodsBase.cs`, 32
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSourcesSizeBalancedTreeMethods.cs`, 35
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksTargetsSizeBalancedTreeMethods.cs`, 36
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSizeBalancedTreeMethodsBase.cs`, 37
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksSourcesSizeBalancedTreeMethods.cs`, 39
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/InternalLinksTargetsSizeBalancedTreeMethods.cs`, 40
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinks.cs`, 41
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/SplitMemoryLinksBase.cs`, 42
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/UnusedLinksListMethods.cs`, 52
`./csharp/Platform.Data.Doublets/Memory/Split/RawLinkDataPart.cs`, 53
`./csharp/Platform.Data.Doublets/Memory/Split/RawLinkIndexPart.cs`, 53
`./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSizeBalancedTreeMethodsBase.cs`, 54
`./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksSourcesSizeBalancedTreeMethods.cs`, 55
`./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32ExternalLinksTargetsSizeBalancedTreeMethods.cs`, 56
`./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSizeBalancedTreeMethodsBase.cs`, 57
`./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksSourcesSizeBalancedTreeMethods.cs`, 58
`./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32InternalLinksTargetsSizeBalancedTreeMethods.cs`, 59
`./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32SplitMemoryLinks.cs`, 60
`./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt32UnusedLinksListMethods.cs`, 62

./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSizeBalancedTreeMethodsBase.cs, 62
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksSourcesSizeBalancedTreeMethods.cs, 63
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64ExternalLinksTargetsSizeBalancedTreeMethods.cs, 64
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSizeBalancedTreeMethodsBase.cs, 65
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksSourcesSizeBalancedTreeMethods.cs, 66
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64InternalLinksTargetsSizeBalancedTreeMethods.cs, 67
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64SplitMemoryLinks.cs, 68
./csharp/Platform.Data.Doublets/Memory/Split/Specific/UInt64UnusedLinksListMethods.cs, 70
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksAvlBalancedTreeMethodsBase.cs, 70
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSizeBalancedTreeMethodsBase.cs, 74
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesAvlBalancedTreeMethods.cs, 78
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesSizeBalancedTreeMethods.cs, 79
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsAvlBalancedTreeMethods.cs, 80
./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsSizeBalancedTreeMethods.cs, 81
./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinks.cs, 82
./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinksBase.cs, 83
./csharp/Platform.Data.Doublets/Memory/United/Generic/UnusedLinksListMethods.cs, 90
./csharp/Platform.Data.Doublets/Memory/United/RawLink.cs, 91
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSizeBalancedTreeMethodsBase.cs, 92
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksSourcesSizeBalancedTreeMethods.cs, 93
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32LinksTargetsSizeBalancedTreeMethods.cs, 94
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32UnitedMemoryLinks.cs, 95
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt32UnusedLinksListMethods.cs, 96
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs, 97
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs, 98
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs, 99
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.cs, 101
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs, 102
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs, 103
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnitedMemoryLinks.cs, 104
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnusedLinksListMethods.cs, 106
./csharp/Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs, 106
./csharp/Platform.Data.Doublets/Numbers/Unary/LinkToToltsFrequencyNumberConveter.cs, 107
./csharp/Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs, 107
./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs, 108
./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs, 109
./csharp/Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 110
./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 110
./csharp/Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 111
./csharp/Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 112
./csharp/Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 115
./csharp/Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 116
./csharp/Platform.Data.Doublets/Sequences/Converters/SequenceToToltsLocalElementLevelsConverter.cs, 117
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs, 118
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs, 118
./csharp/Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 118
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 119
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 120
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 122
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 124
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToltsFrequencyValueConverter.cs, 124
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 125
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 125
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 126
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 126
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 127
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 127
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 128
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 128
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 129
./csharp/Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 129
./csharp/Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 130
./csharp/Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 131
./csharp/Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 131
./csharp/Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 132
./csharp/Platform.Data.Doublets/Sequences/Indexes/Unindex.cs, 132
./csharp/Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 133

- ./csharp/Platform.Data.Doublets/Sequences/Sequences.cs, 160
- ./csharp/Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 171
- ./csharp/Platform.Data.Doublets/Sequences/SequencesOptions.cs, 171
- ./csharp/Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 174
- ./csharp/Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 174
- ./csharp/Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 175
- ./csharp/Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 176
- ./csharp/Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 177
- ./csharp/Platform.Data.Doublets/Stacks/Stack.cs, 178
- ./csharp/Platform.Data.Doublets/Stacks/StackExtensions.cs, 178
- ./csharp/Platform.Data.Doublets/SynchronizedLinks.cs, 179
- ./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs, 180
- ./csharp/Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 182
- ./csharp/Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 187
- ./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 188
- ./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSymbolsListConverter.cs, 189
- ./csharp/Platform.Data.Doublets/Unicode/UnicodeMap.cs, 189
- ./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 192
- ./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 192
- ./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.cs, 193