

LinksPlatform's Platform.Data.Doublets Class Library

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

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

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

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

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

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

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

```

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

```

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

```

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

```

```

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

```

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

```

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

```

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

```

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

```

```

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

```

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

```

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

```

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

```

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

```

```

14         public override TLink Create(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(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
197     ↪ matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<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 = (IList<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 }

```



```

40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     public override bool Equals(object obj) => obj is Doublet<T> doublet ?
        ↳ base.Equals(doublet) : false;
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     public override int GetHashCode() => (Source, Target).GetHashCode();
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     public static bool operator ==(Doublet<T> left, Doublet<T> right) => left.Equals(right);
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     public static bool operator !=(Doublet<T> left, Doublet<T> right) => !(left == right);
52 }
53 }

```

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
            ↳ amountOfCreations)
23         {
24             var random = RandomHelpers.Default;
25             var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
26             var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;

```

```

27     for (var i = OUL; i < amountOfCreations; i++)
28     {
29         var linksAddressRange = new Range<ulong>(0,
30             ↳ addressToUInt64Converter.Convert(links.Count()));
31         var source =
32             ↳ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
33         var target =
34             ↳ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
35         links.GetOrCreate(source, target);
36     }
37 }
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 public static void RunRandomSearches<TLink>(this ILinks<TLink> links, ulong
41     ↳ amountOfSearches)
42 {
43     var random = RandomHelpers.Default;
44     var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
45     var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
46     for (var i = OUL; i < amountOfSearches; i++)
47     {
48         var linksAddressRange = new Range<ulong>(0,
49             ↳ addressToUInt64Converter.Convert(links.Count()));
50         var source =
51             ↳ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
52         var target =
53             ↳ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
54         links.SearchOrDefault(source, target);
55     }
56 }
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, ulong
60     ↳ amountOfDeletions)
61 {
62     var random = RandomHelpers.Default;
63     var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
64     var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
65     var linksCount = addressToUInt64Converter.Convert(links.Count());
66     var min = amountOfDeletions > linksCount ? OUL : linksCount - amountOfDeletions;
67     for (var i = OUL; i < amountOfDeletions; i++)
68     {
69         linksCount = addressToUInt64Converter.Convert(links.Count());
70         if (linksCount <= min)
71         {
72             break;
73         }
74         var linksAddressRange = new Range<ulong>(min, linksCount);
75         var link =
76             ↳ uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
77         links.Delete(link);
78     }
79 }
80
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 public static void Delete<TLink>(this ILinks<TLink> links, TLink linkToDelete) =>
83     ↳ links.Delete(new LinkAddress<TLink>(linkToDelete));
84
85 /// <remarks>
86 /// TODO: Возможно есть очень простой способ это сделать.
87 /// (Например просто удалить файл, или изменить его размер таким образом,
88 /// чтобы удалился весь контент)
89 /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
90 /// </remarks>
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 public static void DeleteAll<TLink>(this ILinks<TLink> links)
93 {
94     var equalityComparer = EqualityComparer<TLink>.Default;
95     var comparer = Comparer<TLink>.Default;
96     for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
97         ↳ Arithmetic.Decrement(i))
98     {
99         links.Delete(i);
100         if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
101         {
102             i = links.Count();
103         }
104     }
105 }

```

```

94     }
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     public static TLink First<TLink>(this ILinks<TLink> links)
98     {
99         TLink firstLink = default;
100         var equalityComparer = EqualityComparer<TLink>.Default;
101         if (equalityComparer.Equals(links.Count(), default))
102         {
103             throw new InvalidOperationException("В хранилище нет связей.");
104         }
105         links.Each(links.Constants.Any, links.Constants.Any, link =>
106         {
107             firstLink = link[links.Constants.IndexPart];
108             return links.Constants.Break;
109         });
110         if (equalityComparer.Equals(firstLink, default))
111         {
112             throw new InvalidOperationException("В процессе поиска по хранилищу не было
113                 ↳ найдено связей.");
114         }
115         return firstLink;
116     }
117
118     #region Paths
119
120     /// <remarks>
121     /// TODO: Как так? Как то что ниже может быть корректно?
122     /// Скорее всего практически не применимо
123     /// Предполагалось, что можно было конвертировать формируемый в проходе через
124     /// ↳ SequenceWalker
125     /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
126     /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
127     /// </remarks>
128     [MethodImpl(MethodImplOptions.AggressiveInlining)]
129     public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
130     ↳ path)
131     {
132         var current = path[0];
133         //EnsureLinkExists(current, "path");
134         if (!links.Exists(current))
135         {
136             return false;
137         }
138         var equalityComparer = EqualityComparer<TLink>.Default;
139         var constants = links.Constants;
140         for (var i = 1; i < path.Length; i++)
141         {
142             var next = path[i];
143             var values = links.GetLink(current);
144             var source = values[constants.SourcePart];
145             var target = values[constants.TargetPart];
146             if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
147                 ↳ next))
148             {
149                 //throw new InvalidOperationException(string.Format("Невозможно выбрать
150                 ↳ путь, так как и Source и Target совпадают с элементом пути {0}.", next));
151                 return false;
152             }
153             if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
154                 ↳ target))
155             {
156                 //throw new InvalidOperationException(string.Format("Невозможно продолжить
157                 ↳ путь через элемент пути {0}", next));
158                 return false;
159             }
160             current = next;
161         }
162         return true;
163     }
164
165     /// <remarks>
166     /// Может потребовать дополнительного стека для PathElement's при использовании
167     /// ↳ SequenceWalker.
168     /// </remarks>
169     [MethodImpl(MethodImplOptions.AggressiveInlining)]
170     public static TLink GetByKey<TLink>(this ILinks<TLink> links, TLink root, params int[]
171     ↳ path)

```

```

163 {
164     links.EnsureLinkExists(root, "root");
165     var currentLink = root;
166     for (var i = 0; i < path.Length; i++)
167     {
168         currentLink = links.GetLink(currentLink)[path[i]];
169     }
170     return currentLink;
171 }
172
173 [MethodImpl(MethodImplOptions.AggressiveInlining)]
174 public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
↪ links, TLink root, ulong size, ulong index)
175 {
176     var constants = links.Constants;
177     var source = constants.SourcePart;
178     var target = constants.TargetPart;
179     if (!Platform.Numbers.Math.IsPowerOfTwo(size))
180     {
181         throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
↪ than powers of two are not supported.");
182     }
183     var path = new BitArray(BitConverter.GetBytes(index));
184     var length = Bit.GetLowestPosition(size);
185     links.EnsureLinkExists(root, "root");
186     var currentLink = root;
187     for (var i = length - 1; i >= 0; i--)
188     {
189         currentLink = links.GetLink(currentLink)[path[i] ? target : source];
190     }
191     return currentLink;
192 }
193
194 #endregion
195
196 /// <summary>
197 /// Возвращает индекс указанной связи.
198 /// </summary>
199 /// <param name="links">Хранилище связей.</param>
200 /// <param name="link">Связь представленная списком, состоящим из её адреса и
↪ содержимого.</param>
201 /// <returns>Индекс начальной связи для указанной связи.</returns>
202 [MethodImpl(MethodImplOptions.AggressiveInlining)]
203 public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
↪ link[links.Constants.IndexPart];
204
205 /// <summary>
206 /// Возвращает индекс начальной (Source) связи для указанной связи.
207 /// </summary>
208 /// <param name="links">Хранилище связей.</param>
209 /// <param name="link">Индекс связи.</param>
210 /// <returns>Индекс начальной связи для указанной связи.</returns>
211 [MethodImpl(MethodImplOptions.AggressiveInlining)]
212 public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
↪ links.GetLink(link)[links.Constants.SourcePart];
213
214 /// <summary>
215 /// Возвращает индекс начальной (Source) связи для указанной связи.
216 /// </summary>
217 /// <param name="links">Хранилище связей.</param>
218 /// <param name="link">Связь представленная списком, состоящим из её адреса и
↪ содержимого.</param>
219 /// <returns>Индекс начальной связи для указанной связи.</returns>
220 [MethodImpl(MethodImplOptions.AggressiveInlining)]
221 public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
↪ link[links.Constants.SourcePart];
222
223 /// <summary>
224 /// Возвращает индекс конечной (Target) связи для указанной связи.
225 /// </summary>
226 /// <param name="links">Хранилище связей.</param>
227 /// <param name="link">Индекс связи.</param>
228 /// <returns>Индекс конечной связи для указанной связи.</returns>
229 [MethodImpl(MethodImplOptions.AggressiveInlining)]
230 public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
↪ links.GetLink(link)[links.Constants.TargetPart];
231
232 /// <summary>

```



```

233 /// Возвращает индекс конечной (Target) связи для указанной связи.
234 /// </summary>
235 /// <param name="links">Хранилище связей.</param>
236 /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↳ содержимого.</param>
237 /// <returns>Индекс конечной связи для указанной связи.</returns>
238 [MethodImpl(MethodImplOptions.AggressiveInlining)]
239 public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↳ link[links.Constants.TargetPart];
240
241 /// <summary>
242 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
243 /// </summary>
244 /// <param name="links">Хранилище связей.</param>
245 /// <param name="handler">Обработчик каждой подходящей связи.</param>
246 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
    ↳ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
    ↳ Any - отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
247 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
248 [MethodImpl(MethodImplOptions.AggressiveInlining)]
249 public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
    ↳ handler, params TLink[] restrictions)
250     => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
    ↳ links.Constants.Continue);
251
252 /// <summary>
253 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
254 /// </summary>
255 /// <param name="links">Хранилище связей.</param>
256 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
    ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
257 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
    ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
258 /// <param name="handler">Обработчик каждой подходящей связи.</param>
259 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
260 [MethodImpl(MethodImplOptions.AggressiveInlining)]
261 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
    ↳ Func<TLink, bool> handler)
262 {
263     var constants = links.Constants;
264     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
    ↳ constants.Break, constants.Any, source, target);
265 }
266
267 /// <summary>
268 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
269 /// </summary>
270 /// <param name="links">Хранилище связей.</param>
271 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
    ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
272 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
    ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
273 /// <param name="handler">Обработчик каждой подходящей связи.</param>
274 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
275 [MethodImpl(MethodImplOptions.AggressiveInlining)]
276 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
    ↳ Func<IList<TLink>, TLink> handler) => links.Each(handler, links.Constants.Any,
    ↳ source, target);
277
278 [MethodImpl(MethodImplOptions.AggressiveInlining)]
279 public static IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ restrictions)
280 {
281     var arraySize = CheckedConverter<TLink,
    ↳ long>.Default.Convert(links.Count(restrictions));
282     if (arraySize > 0)

```

```

283     {
284         var array = new IList<TLink>[arraySize];
285         var filler = new ArrayFiller<IList<TLink>, TLink>(array,
286             ↪ links.Constants.Continue);
287         links.Each(filler.AddAndReturnConstant, restrictions);
288         return array;
289     }
290     else
291     {
292         return Array.Empty<IList<TLink>>();
293     }
294 }
295
296 [MethodImpl(MethodImplOptions.AggressiveInlining)]
297 public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
298     ↪ restrictions)
299 {
300     var arraySize = CheckedConverter<TLink,
301         ↪ long>.Default.Convert(links.Count(restrictions));
302     if (arraySize > 0)
303     {
304         var array = new TLink[arraySize];
305         var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
306         links.Each(filler.AddFirstAndReturnConstant, restrictions);
307         return array;
308     }
309     else
310     {
311         return Array.Empty<TLink>();
312     }
313 }
314
315 /// <summary>
316 /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
317 ↪ в хранилище связей.
318 /// </summary>
319 /// <param name="links">Хранилище связей.</param>
320 /// <param name="source">Начало связи.</param>
321 /// <param name="target">Конец связи.</param>
322 /// <returns>Значение, определяющее существует ли связь.</returns>
323 [MethodImpl(MethodImplOptions.AggressiveInlining)]
324 public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
325     ↪ => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
326     ↪ default) > 0;
327
328 #region Ensure
329 // TODO: May be move to EnsureExtensions or make it both there and here
330
331 [MethodImpl(MethodImplOptions.AggressiveInlining)]
332 public static void EnsureLinkExists<TLink>(this ILinks<TLink> links, IList<TLink>
333     ↪ restrictions)
334 {
335     for (var i = 0; i < restrictions.Count; i++)
336     {
337         if (!links.Exists(restrictions[i]))
338         {
339             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
340                 ↪ $"sequence[{i}]");
341         }
342     }
343 }
344
345 [MethodImpl(MethodImplOptions.AggressiveInlining)]
346 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
347     ↪ reference, string argumentName)
348 {
349     if (links.Constants.IsInternalReference(reference) && !links.Exists(reference))
350     {
351         throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
352     }
353 }
354
355 [MethodImpl(MethodImplOptions.AggressiveInlining)]
356 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
357     ↪ IList<TLink> restrictions, string argumentName)
358 {
359     for (int i = 0; i < restrictions.Count; i++)
360     {

```

```

351         links.EnsureInnerReferenceExists(restrictions[i], argumentName);
352     }
353 }
354
355 [MethodImpl(MethodImplOptions.AggressiveInlining)]
356 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
    ↪ restrictions)
357 {
358     var equalityComparer = EqualityComparer<TLink>.Default;
359     var any = links.Constants.Any;
360     for (var i = 0; i < restrictions.Count; i++)
361     {
362         if (!equalityComparer.Equals(restrictions[i], any) &&
            ↪ !links.Exists(restrictions[i]))
363         {
364             throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                ↪ $"sequence[{i}]");
365         }
366     }
367 }
368
369 [MethodImpl(MethodImplOptions.AggressiveInlining)]
370 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
    ↪ string argumentName)
371 {
372     var equalityComparer = EqualityComparer<TLink>.Default;
373     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
374     {
375         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
376     }
377 }
378
379 [MethodImpl(MethodImplOptions.AggressiveInlining)]
380 public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
    ↪ link, string argumentName)
381 {
382     var equalityComparer = EqualityComparer<TLink>.Default;
383     if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
384     {
385         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
386     }
387 }
388
389 /// <param name="links">Хранилище связей.</param>
390 [MethodImpl(MethodImplOptions.AggressiveInlining)]
391 public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
    ↪ TLink target)
392 {
393     if (links.Exists(source, target))
394     {
395         throw new LinkWithSameValueAlreadyExistsException();
396     }
397 }
398
399 /// <param name="links">Хранилище связей.</param>
400 [MethodImpl(MethodImplOptions.AggressiveInlining)]
401 public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
402 {
403     if (links.HasUsages(link))
404     {
405         throw new ArgumentLinkHasDependenciesException<TLink>(link);
406     }
407 }
408
409 /// <param name="links">Хранилище связей.</param>
410 [MethodImpl(MethodImplOptions.AggressiveInlining)]
411 public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↪ addresses) => links.EnsureCreated(links.Create, addresses);
412
413 /// <param name="links">Хранилище связей.</param>
414 [MethodImpl(MethodImplOptions.AggressiveInlining)]
415 public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↪ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
416
417 /// <param name="links">Хранилище связей.</param>
418 [MethodImpl(MethodImplOptions.AggressiveInlining)]
419 public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
    ↪ params TLink[] addresses)

```

```

420 {
421     var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
422     var uint64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
423     var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
424         ↪ !links.Exists(x)));
425     if (nonExistentAddresses.Count > 0)
426     {
427         var max = nonExistentAddresses.Max();
428         max = uint64ToAddressConverter.Convert(System.Math.Min(addressToUInt64Converter.
429             ↪ Convert(max),
430             ↪ addressToUInt64Converter.Convert(links.Constants.InternalReferencesRange.Max
431             ↪ imum)));
432         var createdLinks = new List<TLink>();
433         var equalityComparer = EqualityComparer<TLink>.Default;
434         TLink createdLink = creator();
435         while (!equalityComparer.Equals(createdLink, max))
436         {
437             createdLinks.Add(createdLink);
438         }
439         for (var i = 0; i < createdLinks.Count; i++)
440         {
441             if (!nonExistentAddresses.Contains(createdLinks[i]))
442             {
443                 links.Delete(createdLinks[i]);
444             }
445         }
446     }
447 }
448
449 #endregion
450
451 /// <param name="links">Хранилище связей.</param>
452 [MethodImpl(MethodImplOptions.AggressiveInlining)]
453 public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
454 {
455     var constants = links.Constants;
456     var values = links.GetLink(link);
457     TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link,
458         ↪ constants.Any));
459     var equalityComparer = EqualityComparer<TLink>.Default;
460     if (equalityComparer.Equals(values[constants.SourcePart], link))
461     {
462         usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
463     }
464     TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
465         ↪ link));
466     if (equalityComparer.Equals(values[constants.TargetPart], link))
467     {
468         usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
469     }
470     return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
471 }
472
473 /// <param name="links">Хранилище связей.</param>
474 [MethodImpl(MethodImplOptions.AggressiveInlining)]
475 public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
476     ↪ Comparer<TLink>.Default.Compare(links.CountUsages(link), default) > 0;
477
478 /// <param name="links">Хранилище связей.</param>
479 [MethodImpl(MethodImplOptions.AggressiveInlining)]
480 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
481     ↪ TLink target)
482 {
483     var constants = links.Constants;
484     var values = links.GetLink(link);
485     var equalityComparer = EqualityComparer<TLink>.Default;
486     return equalityComparer.Equals(values[constants.SourcePart], source) &&
487         ↪ equalityComparer.Equals(values[constants.TargetPart], target);
488 }
489
490 /// <summary>
491 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
492 /// </summary>
493 /// <param name="links">Хранилище связей.</param>
494 /// <param name="source">Индекс связи, которая является началом для искомой
495     ↪ связи.</param>
496 /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>

```

```

487  /// <returns>Индекс искомой связи с указанными Source (началом) и Target
488  ↪ (концом).</returns>
489  [MethodImpl(MethodImplOptions.AggressiveInlining)]
490  public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
491  ↪ target)
492  {
493      var constants = links.Constants;
494      var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break, default);
495      links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
496      return setter.Result;
497  }
498
499  /// <param name="links">Хранилище связей.</param>
500  [MethodImpl(MethodImplOptions.AggressiveInlining)]
501  public static TLink Create<TLink>(this ILinks<TLink> links) => links.Create(null);
502
503  /// <param name="links">Хранилище связей.</param>
504  [MethodImpl(MethodImplOptions.AggressiveInlining)]
505  public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
506  {
507      var link = links.Create();
508      return links.Update(link, link, link);
509  }
510
511  /// <param name="links">Хранилище связей.</param>
512  [MethodImpl(MethodImplOptions.AggressiveInlining)]
513  public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
514  ↪ target) => links.Update(links.Create(), source, target);
515
516  /// <summary>
517  /// Обновляет связь с указанными началом (Source) и концом (Target)
518  /// на связь с указанными началом (NewSource) и концом (NewTarget).
519  /// </summary>
520  /// <param name="links">Хранилище связей.</param>
521  /// <param name="link">Индекс обновляемой связи.</param>
522  /// <param name="newSource">Индекс связи, которая является началом связи, на которую
523  ↪ выполняется обновление.</param>
524  /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
525  ↪ выполняется обновление.</param>
526  /// <returns>Индекс обновлённой связи.</returns>
527  [MethodImpl(MethodImplOptions.AggressiveInlining)]
528  public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
529  ↪ TLink newTarget) => links.Update(new LinkAddress<TLink>(link), new Link<TLink>(link,
530  ↪ newSource, newTarget));
531
532  /// <summary>
533  /// Обновляет связь с указанными началом (Source) и концом (Target)
534  /// на связь с указанными началом (NewSource) и концом (NewTarget).
535  /// </summary>
536  /// <param name="links">Хранилище связей.</param>
537  /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
538  ↪ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
539  ↪ Itself - требование установить ссылку на себя, 1..∞ конкретный адрес другой
540  ↪ связи.</param>
541  /// <returns>Индекс обновлённой связи.</returns>
542  [MethodImpl(MethodImplOptions.AggressiveInlining)]
543  public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
544  {
545      if (restrictions.Length == 2)
546      {
547          return links.MergeAndDelete(restrictions[0], restrictions[1]);
548      }
549      if (restrictions.Length == 4)
550      {
551          return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
552          ↪ restrictions[2], restrictions[3]);
553      }
554      else
555      {
556          return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
557      }
558  }
559
560  [MethodImpl(MethodImplOptions.AggressiveInlining)]
561  public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
562  ↪ links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
563  {
564      var equalityComparer = EqualityComparer<TLink>.Default;

```

```

553     var constants = links.Constants;
554     var restrictionsIndex = restrictions[constants.IndexPart];
555     var substitutionIndex = substitution[constants.IndexPart];
556     if (equalityComparer.Equals(substitutionIndex, default))
557     {
558         substitutionIndex = restrictionsIndex;
559     }
560     var source = substitution[constants.SourcePart];
561     var target = substitution[constants.TargetPart];
562     source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
563     target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
564     return new Link<TLink>(substitutionIndex, source, target);
565 }
566
567 /// <summary>
568 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
569   ↳ с указанными Source (началом) и Target (концом).
570 /// </summary>
571 /// <param name="links">Хранилище связей.</param>
572 /// <param name="source">Индекс связи, которая является началом на создаваемой
573   ↳ связи.</param>
574 /// <param name="target">Индекс связи, которая является концом для создаваемой
575   ↳ связи.</param>
576 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
577 [MethodImpl(MethodImplOptions.AggressiveInlining)]
578 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
579   ↳ target)
580 {
581     var link = links.SearchOrDefault(source, target);
582     if (EqualityComparer<TLink>.Default.Equals(link, default))
583     {
584         link = links.CreateAndUpdate(source, target);
585     }
586     return link;
587 }
588
589 /// <summary>
590 /// Обновляет связь с указанными началом (Source) и концом (Target)
591   ↳ на связь с указанными началом (NewSource) и концом (NewTarget).
592 /// </summary>
593 /// <param name="links">Хранилище связей.</param>
594 /// <param name="source">Индекс связи, которая является началом обновляемой
595   ↳ связи.</param>
596 /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
597 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
598   ↳ выполняется обновление.</param>
599 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
600   ↳ выполняется обновление.</param>
601 /// <returns>Индекс обновлённой связи.</returns>
602 [MethodImpl(MethodImplOptions.AggressiveInlining)]
603 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
604   ↳ TLink target, TLink newSource, TLink newTarget)
605 {
606     var equalityComparer = EqualityComparer<TLink>.Default;
607     var link = links.SearchOrDefault(source, target);
608     if (equalityComparer.Equals(link, default))
609     {
610         return links.CreateAndUpdate(newSource, newTarget);
611     }
612     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
613   ↳ target))
614     {
615         return link;
616     }
617     return links.Update(link, newSource, newTarget);
618 }
619
620 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
621 /// <param name="links">Хранилище связей.</param>
622 /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
623 /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
624 [MethodImpl(MethodImplOptions.AggressiveInlining)]
625 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
626   ↳ target)
627 {
628     var link = links.SearchOrDefault(source, target);
629     if (!EqualityComparer<TLink>.Default.Equals(link, default))

```

```

620     {
621         links.Delete(link);
622         return link;
623     }
624     return default;
625 }
626
627 /// <summary>Удаляет несколько связей.</summary>
628 /// <param name="links">Хранилище связей.</param>
629 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
630 [MethodImpl(MethodImplOptions.AggressiveInlining)]
631 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
632 {
633     for (int i = 0; i < deletedLinks.Count; i++)
634     {
635         links.Delete(deletedLinks[i]);
636     }
637 }
638
639 /// <remarks>Before execution of this method ensure that deleted link is detached (all
640   ↳ values - source and target are reset to null) or it might enter into infinite
641   ↳ recursion.</remarks>
642 [MethodImpl(MethodImplOptions.AggressiveInlining)]
643 public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
644 {
645     var anyConstant = links.Constants.Any;
646     var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
647     links.DeleteByQuery(usagesAsSourceQuery);
648     var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
649     links.DeleteByQuery(usagesAsTargetQuery);
650 }
651
652 [MethodImpl(MethodImplOptions.AggressiveInlining)]
653 public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
654 {
655     var count = CheckedConverter<TLink, long>.Default.Convert(links.Count(query));
656     if (count > 0)
657     {
658         var queryResult = new TLink[count];
659         var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
660   ↳ links.Constants.Continue);
661         links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
662         for (var i = count - 1; i >= 0; i--)
663         {
664             links.Delete(queryResult[i]);
665         }
666     }
667 }
668
669 // TODO: Move to Platform.Data
670 [MethodImpl(MethodImplOptions.AggressiveInlining)]
671 public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
672 {
673     var nullConstant = links.Constants.Null;
674     var equalityComparer = EqualityComparer<TLink>.Default;
675     var link = links.GetLink(linkIndex);
676     for (int i = 1; i < link.Count; i++)
677     {
678         if (!equalityComparer.Equals(link[i], nullConstant))
679         {
680             return false;
681         }
682     }
683     return true;
684 }
685
686 // TODO: Create a universal version of this method in Platform.Data (with using of for
687   ↳ loop)
688 [MethodImpl(MethodImplOptions.AggressiveInlining)]
689 public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
690 {
691     var nullConstant = links.Constants.Null;
692     var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
693     links.Update(updateRequest);
694 }
695
696 // TODO: Create a universal version of this method in Platform.Data (with using of for
697   ↳ loop)

```

```

693 [MethodImpl(MethodImplOptions.AggressiveInlining)]
694 public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
695 {
696     if (!links.AreValuesReset(linkIndex))
697     {
698         links.ResetValues(linkIndex);
699     }
700 }
701
702 /// <summary>
703 /// Merging two usages graphs, all children of old link moved to be children of new link
704   ↳ or deleted.
705 /// </summary>
706 [MethodImpl(MethodImplOptions.AggressiveInlining)]
707 public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
708   ↳ TLink newLinkIndex)
709 {
710     var addressToInt64Converter = CheckedConverter<TLink, long>.Default;
711     var equalityComparer = EqualityComparer<TLink>.Default;
712     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
713     {
714         var constants = links.Constants;
715         var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
716   ↳ constants.Any);
717         var usagesAsSourceCount =
718   ↳ addressToInt64Converter.Convert(links.Count(usagesAsSourceQuery));
719         var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
720   ↳ oldLinkIndex);
721         var usagesAsTargetCount =
722   ↳ addressToInt64Converter.Convert(links.Count(usagesAsTargetQuery));
723         var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
724   ↳ usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
725         if (!isStandalonePoint)
726         {
727             var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
728             if (totalUsages > 0)
729             {
730                 var usages = ArrayPool.Allocate<TLink>(totalUsages);
731                 var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
732   ↳ links.Constants.Continue);
733                 var i = 0L;
734                 if (usagesAsSourceCount > 0)
735                 {
736                     links.Each(usagesFiller.AddFirstAndReturnConstant,
737   ↳ usagesAsSourceQuery);
738                     for (; i < usagesAsSourceCount; i++)
739                     {
740                         var usage = usages[i];
741                         if (!equalityComparer.Equals(usage, oldLinkIndex))
742                         {
743                             links.Update(usage, newLinkIndex, links.GetTarget(usage));
744                         }
745                     }
746                 }
747                 if (usagesAsTargetCount > 0)
748                 {
749                     links.Each(usagesFiller.AddFirstAndReturnConstant,
750   ↳ usagesAsTargetQuery);
751                     for (; i < usages.Length; i++)
752                     {
753                         var usage = usages[i];
754                         if (!equalityComparer.Equals(usage, oldLinkIndex))
755                         {
756                             links.Update(usage, links.GetSource(usage), newLinkIndex);
757                         }
758                     }
759                 }
760                 ArrayPool.Free(usages);
761             }
762         }
763     }
764     return newLinkIndex;
765 }
766
767 /// <summary>
768 /// Replace one link with another (replaced link is deleted, children are updated or
769   ↳ deleted).
770 /// </summary>

```



```

760 [MethodImpl(MethodImplOptions.AggressiveInlining)]
761 public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
    ↪ TLink newLinkIndex)
762 {
763     var equalityComparer = EqualityComparer<TLink>.Default;
764     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
765     {
766         links.MergeUsages(oldLinkIndex, newLinkIndex);
767         links.Delete(oldLinkIndex);
768     }
769     return newLinkIndex;
770 }
771
772 [MethodImpl(MethodImplOptions.AggressiveInlining)]
773 public static ILinks<TLink>
    ↪ DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
774 {
775     links = new LinksCascadeUsagesResolver<TLink>(links);
776     links = new NonNullContentsLinkDeletionResolver<TLink>(links);
777     links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
778     return links;
779 }
780
781 [MethodImpl(MethodImplOptions.AggressiveInlining)]
782 public static string Format<TLink>(this ILinks<TLink> links, IList<TLink> link)
783 {
784     var constants = links.Constants;
785     return $"({link[constants.IndexPart]}: {link[constants.SourcePart]}
    ↪ {link[constants.TargetPart]});";
786 }
787
788 [MethodImpl(MethodImplOptions.AggressiveInlining)]
789 public static string Format<TLink>(this ILinks<TLink> links, TLink link) =>
    ↪ links.Format(links.GetLink(link));
790 }
791 }

```

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>,
    ↪ LinksConstants<TLink>>, ILinks<TLink>
6      {
7      }
8  }

```

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 =
    ↪ EqualityComparer<TLink>.Default;
12
13         private readonly TLink _frequencyMarker;
14         private readonly TLink _unaryOne;
15         private readonly IIncrementer<TLink> _unaryNumberIncrementer;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
    ↪ IIncrementer<TLink> unaryNumberIncrementer)
19             : base(links)
20         {
21             _frequencyMarker = frequencyMarker;
22             _unaryOne = unaryOne;
23             _unaryNumberIncrementer = unaryNumberIncrementer;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Increment(TLink frequency)
28         {

```

```

29     var links = _links;
30     if (_equalityComparer.Equals(frequency, default))
31     {
32         return links.GetOrCreate(_unaryOne, _frequencyMarker);
33     }
34     var incrementedSource =
35         ↪ _unaryNumberIncrementer.Increment(links.GetSource(frequency));
36     return links.GetOrCreate(incrementedSource, _frequencyMarker);
37 }
38 }

```

1.22 ./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 }

```

```

24     private const int Length = 3;
25
26     public readonly TLink Index;
27     public readonly TLink Source;
28     public readonly TLink Target;
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     public Link(params TLink[] values) => SetValues(values, out Index, out Source, out
        ↪ Target);
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     public Link(ICollection<TLink> values) => SetValues(values, out Index, out Source, out Target);
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     public Link(object other)
38     {
39         if (other is Link<TLink> otherLink)
40         {
41             SetValues(ref otherLink, out Index, out Source, out Target);
42         }
43         else if (other is ICollection<TLink> otherList)
44         {
45             SetValues(otherList, out Index, out Source, out Target);
46         }
47         else
48         {
49             throw new NotSupportedException();
50         }
51     }
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     public Link(ref Link<TLink> other) => SetValues(ref other, out Index, out Source, out
        ↪ Target);
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     public Link(TLink index, TLink source, TLink target)
58     {
59         Index = index;
60         Source = source;
61         Target = target;
62     }
63
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     private static void SetValues(ref Link<TLink> other, out TLink index, out TLink source,
        ↪ out TLink target)
66     {
67         index = other.Index;
68         source = other.Source;
69         target = other.Target;
70     }
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     private static void SetValues(ICollection<TLink> values, out TLink index, out TLink source,
        ↪ out TLink target)
74     {
75         switch (values.Count)
76         {
77             case 3:
78                 index = values[0];
79                 source = values[1];
80                 target = values[2];
81                 break;
82             case 2:
83                 index = values[0];
84                 source = values[1];
85                 target = default;
86                 break;
87             case 1:
88                 index = values[0];
89                 source = default;
90                 target = default;
91                 break;
92             default:
93                 index = default;
94                 source = default;
95                 target = default;
96                 break;
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> &&
    ↳ Equals((Link<TLink>)other);
110
111 [MethodImpl(MethodImplOptions.AggressiveInlining)]
112 public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
113     && _equalityComparer.Equals(Source, other.Source)
114     && _equalityComparer.Equals(Target, other.Target);
115
116 [MethodImpl(MethodImplOptions.AggressiveInlining)]
117 public static string ToString(TLink index, TLink source, TLink target) => $"{({index}:
    ↳ {source}->{target})}";
118
119 [MethodImpl(MethodImplOptions.AggressiveInlining)]
120 public static string ToString(TLink source, TLink target) => $"{({source}->{target})}";
121
122 [MethodImpl(MethodImplOptions.AggressiveInlining)]
123 public static implicit operator TLink[] (Link<TLink> link) => link.ToArray();
124
125 [MethodImpl(MethodImplOptions.AggressiveInlining)]
126 public static implicit operator Link<TLink>(TLink[] linkArray) => new
    ↳ Link<TLink>(linkArray);
127
128 [MethodImpl(MethodImplOptions.AggressiveInlining)]
129 public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
    ↳ ToString(Source, Target) : ToString(Index, Source, Target);
130
131 #region IList
132
133 public int Count
134 {
135     [MethodImpl(MethodImplOptions.AggressiveInlining)]
136     get => Length;
137 }
138
139 public bool IsReadOnly
140 {
141     [MethodImpl(MethodImplOptions.AggressiveInlining)]
142     get => true;
143 }
144
145 public TLink this[int index]
146 {
147     [MethodImpl(MethodImplOptions.AggressiveInlining)]
148     get
149     {
150         Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
            ↳ nameof(index));
151         if (index == _constants.IndexPart)
152         {
153             return Index;
154         }
155         if (index == _constants.SourcePart)
156         {
157             return Source;
158         }
159         if (index == _constants.TargetPart)
160         {
161             return Target;
162         }
163         throw new NotSupportedException(); // Impossible path due to
            ↳ 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),
194         ↪ nameof(arrayIndex));
195     if (arrayIndex + Length > array.Length)
196     {
197         throw new InvalidOperationException();
198     }
199     array[arrayIndex++] = Index;
200     array[arrayIndex++] = Source;
201     array[arrayIndex] = Target;
202 }
203
204 [MethodImpl(MethodImplOptions.AggressiveInlining)]
205 public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
206
207 [MethodImpl(MethodImplOptions.AggressiveInlining)]
208 public int IndexOf(TLink item)
209 {
210     if (_equalityComparer.Equals(Index, item))
211     {
212         return _constants.IndexPart;
213     }
214     if (_equalityComparer.Equals(Source, item))
215     {
216         return _constants.SourcePart;
217     }
218     if (_equalityComparer.Equals(Target, item))
219     {
220         return _constants.TargetPart;
221     }
222     return -1;
223 }
224
225 [MethodImpl(MethodImplOptions.AggressiveInlining)]
226 public void Insert(int index, TLink item) => throw new NotSupportedException();
227
228 [MethodImpl(MethodImplOptions.AggressiveInlining)]
229 public void RemoveAt(int index) => throw new NotSupportedException();
230
231 [MethodImpl(MethodImplOptions.AggressiveInlining)]
232 public static bool operator ==(Link<TLink> left, Link<TLink> right) =>
233     ↪ left.Equals(right);
234
235 [MethodImpl(MethodImplOptions.AggressiveInlining)]
236 public static bool operator !=(Link<TLink> left, Link<TLink> right) => !(left == right);
237
238 #endregion
239 }
240 }

```

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         [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 }
```

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

```



```

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

```

```

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

```

```

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

```

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

```

```
56     }
57 }
```

1.32 ./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> :
8     ↪ ExternalLinksSizeBalancedTreeMethodsBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public ExternalLinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants,
12         ↪ byte* linksDataParts, byte* linksIndexParts, byte* header) : base(constants,
13         ↪ linksDataParts, linksIndexParts, header) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected override ref TLink GetLeftReference(TLink node) => ref
17         ↪ GetLinkIndexPartReference(node).LeftAsTarget;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected override ref TLink GetRightReference(TLink node) => ref
21         ↪ GetLinkIndexPartReference(node).RightAsTarget;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected override TLink GetLeft(TLink node) =>
25         ↪ GetLinkIndexPartReference(node).LeftAsTarget;
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected override TLink GetRight(TLink node) =>
29         ↪ GetLinkIndexPartReference(node).RightAsTarget;
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected override void SetLeft(TLink node, TLink left) =>
33         ↪ GetLinkIndexPartReference(node).LeftAsTarget = left;
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override void SetRight(TLink node, TLink right) =>
37         ↪ GetLinkIndexPartReference(node).RightAsTarget = right;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override TLink GetSize(TLink node) =>
41         ↪ GetLinkIndexPartReference(node).SizeAsTarget;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override void SetSize(TLink node, TLink size) =>
45         ↪ GetLinkIndexPartReference(node).SizeAsTarget = size;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override TLink GetBasePartValue(TLink link) =>
52         ↪ GetLinkDataPartReference(link).Target;
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
56         ↪ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
57         ↪ AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource);
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
61         ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
62         ↪ AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource);
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected override void ClearNode(TLink node)
66         {
67             ref var link = ref GetLinkIndexPartReference(node);
68             link.LeftAsTarget = Zero;
69             link.RightAsTarget = Zero;
70             link.SizeAsTarget = Zero;
71         }
72     }
73 }
```

1.33 ./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)
65         {
66             ref var link = ref GetLinkDataPartReference(linkIndex);
67             return new Link<TLink>(linkIndex, link.Source, link.Target);
68         }
69
70         public TLink this[TLink link, TLink index]
71         {
72             [MethodImpl(MethodImplOptions.AggressiveInlining)]
73             get
74             {
75                 var root = GetTreeRoot(link);
76                 if (GreaterOrEqualThan(index, GetSize(root)))
77                 {
78                     return Zero;
79                 }
80             }
81         }
82     }
83 }

```

```

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

```

```

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

```

1.34 ./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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override void SetSize(TLink node, TLink size) =>
45             ↪ GetLinkIndexPartReference(node).SizeAsSource = size;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override TLink GetTreeRoot(TLink link) =>
49             ↪ GetLinkIndexPartReference(link).RootAsSource;
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override TLink GetBasePartValue(TLink link) =>
53             ↪ GetLinkDataPartReference(link).Source;
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

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

```



```

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

```

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

```

```

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;
45         protected readonly long _indexMemoryReservationStepInBytes;
46
47         protected ILinksTreeMethods<TLink> InternalSourcesTreeMethods;
48         protected ILinksTreeMethods<TLink> ExternalSourcesTreeMethods;
49         protected ILinksTreeMethods<TLink> InternalTargetsTreeMethods;
50         protected ILinksTreeMethods<TLink> ExternalTargetsTreeMethods;
51         // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
52         ↳ нужно использовать не список а дерево, так как так можно быстрее проверить на
53         ↳ наличие связи внутри
54         protected ILinksListMethods<TLink> UnusedLinksListMethods;
55
56         /// <summary>
57         /// Возвращает общее число связей находящихся в хранилище.
58         /// </summary>

```

```

53     protected virtual TLink Total
54     {
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         get
57         {
58             ref var header = ref GetHeaderReference();
59             return Subtract(header.AllocatedLinks, header.FreeLinks);
60         }
61     }
62
63     public virtual LinksConstants<TLink> Constants
64     {
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         get;
67     }
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
71     ↪ indexMemory, long memoryReservationStep, LinksConstants<TLink> constants)
72     {
73         _dataMemory = dataMemory;
74         _indexMemory = indexMemory;
75         _dataMemoryReservationStepInBytes = memoryReservationStep * LinkDataPartSizeInBytes;
76         _indexMemoryReservationStepInBytes = memoryReservationStep *
77         ↪ LinkIndexPartSizeInBytes;
78         Constants = constants;
79     }
80
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     protected SplitMemoryLinksBase(IResizableDirectMemory dataMemory, IResizableDirectMemory
83     ↪ indexMemory, long memoryReservationStep) : this(dataMemory, indexMemory,
84     ↪ memoryReservationStep, Default<LinksConstants<TLink>>.Instance) { }
85
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     protected virtual void Init(IResizableDirectMemory dataMemory, IResizableDirectMemory
88     ↪ indexMemory, long memoryReservationStep)
89     {
90         if (dataMemory.ReservedCapacity < memoryReservationStep)
91         {
92             dataMemory.ReservedCapacity = memoryReservationStep;
93         }
94         if (indexMemory.ReservedCapacity < memoryReservationStep)
95         {
96             indexMemory.ReservedCapacity = memoryReservationStep;
97         }
98         SetPointers(dataMemory, indexMemory);
99         ref var header = ref GetHeaderReference();
100         // Ensure correctness _memory.UsedCapacity over _header->AllocatedLinks
101         // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
102         dataMemory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) *
103         ↪ LinkDataPartSizeInBytes + LinkDataPartSizeInBytes; // First link is read only
104         ↪ zero link.
105         indexMemory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) *
106         ↪ LinkIndexPartSizeInBytes + LinkHeaderSizeInBytes;
107         // Ensure correctness _memory.ReservedLinks over _header->ReservedCapacity
108         // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
109         header.ReservedLinks = ConvertToAddress((dataMemory.ReservedCapacity -
110         ↪ LinkDataPartSizeInBytes) / LinkDataPartSizeInBytes);
111     }
112
113     [MethodImpl(MethodImplOptions.AggressiveInlining)]
114     public virtual TLink Count(IList<TLink> restrictions)
115     {
116         // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
117         if (restrictions.Count == 0)
118         {
119             return Total;
120         }
121         var constants = Constants;
122         var any = constants.Any;
123         var index = restrictions[constants.IndexPart];
124         if (restrictions.Count == 1)
125         {
126             if (AreEqual(index, any))
127             {
128                 return Total;
129             }
130             return Exists(index) ? GetOne() : GetZero();
131         }
132     }

```

```

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         else if (AreEqual(target, any))
188         {
189             if (externalReferencesRange.HasValue &&
190                 ⇨ externalReferencesRange.Value.Contains(source))
191             {
192                 return ExternalSourcesTreeMethods.CountUsages(source);
193             }
194             else
195             {
196                 return InternalSourcesTreeMethods.CountUsages(source);
197             }
198         }
199     }
200     else //if(source != Any && target != Any)
201     {

```

```

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

```

```

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,
283             ↳ GetHeaderReference().AllocatedLinks); link = Increment(link))
284         {
285             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
286             {
287                 return @break;
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             if (GreaterThan(InternalSourcesTreeMethods.CountUsages(source),
409                 ↪ InternalTargetsTreeMethods.CountUsages(target)))
410             {
411                 link = InternalTargetsTreeMethods.Search(source, target);
412             }
413             else
414             {
415                 link = InternalSourcesTreeMethods.Search(source, target);
416             }
417         }
418         return AreEqual(link, constants.Null) ? @continue :
419             ↪ handler(GetLinkStruct(link));
420     }
421 }

```

```

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
    ↳ поддерживаются.");
454 }
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))
479         {
480             ExternalSourcesTreeMethods.Detach(ref rootAsSource, linkIndex);
481         }
482         else
483         {
484             InternalSourcesTreeMethods.Detach(ref
485                 ↳ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
486         }
487     }
488 }
489 if (!AreEqual(target, @null))
490 {
491     if (externalReferencesRange.HasValue &&
492         ↳ externalReferencesRange.Value.Contains(target))
493     {

```



```

488         ExternalTargetsTreeMethods.Detach(ref rootAsTarget, linkIndex);
489     }
490     else
491     {
492         InternalTargetsTreeMethods.Detach(ref
493             ↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
494     }
495     source = link.Source = substitution[constants.SourcePart];
496     target = link.Target = substitution[constants.TargetPart];
497     if (!AreEqual(source, @null))
498     {
499         if (externalReferencesRange.HasValue &&
500             ↪ externalReferencesRange.Value.Contains(source))
501         {
502             ExternalSourcesTreeMethods.Attach(ref rootAsSource, linkIndex);
503         }
504         else
505         {
506             InternalSourcesTreeMethods.Attach(ref
507                 ↪ GetLinkIndexPartReference(source).RootAsSource, linkIndex);
508         }
509     }
510     if (!AreEqual(target, @null))
511     {
512         if (externalReferencesRange.HasValue &&
513             ↪ externalReferencesRange.Value.Contains(target))
514         {
515             ExternalTargetsTreeMethods.Attach(ref rootAsTarget, linkIndex);
516         }
517         else
518         {
519             InternalTargetsTreeMethods.Attach(ref
520                 ↪ GetLinkIndexPartReference(target).RootAsTarget, linkIndex);
521         }
522     }
523     return linkIndex;
524 }
525
526 /// <remarks>
527 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
528 ↪ пространство
529 /// </remarks>
530 [MethodImpl(MethodImplOptions.AggressiveInlining)]
531 public virtual TLink Create(ICollection<TLink> restrictions)
532 {
533     ref var header = ref GetHeaderReference();
534     var freeLink = header.FirstFreeLink;
535     if (!AreEqual(freeLink, Constants.Null))
536     {
537         UnusedLinksListMethods.Detach(freeLink);
538     }
539     else
540     {
541         var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
542         if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
543         {
544             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
545         }
546         if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
547         {
548             _dataMemory.ReservedCapacity += _dataMemory.ReservationStepInBytes;
549             _indexMemory.ReservedCapacity += _indexMemory.ReservationStepInBytes;
550             SetPointers(_dataMemory, _indexMemory);
551             header = ref GetHeaderReference();
552             header.ReservedLinks = ConvertToAddress(_dataMemory.ReservedCapacity /
553                 ↪ LinkDataPartSizeInBytes);
554         }
555         header.AllocatedLinks = Increment(header.AllocatedLinks);
556         _dataMemory.UsedCapacity += LinkDataPartSizeInBytes;
557         _indexMemory.UsedCapacity += LinkIndexPartSizeInBytes;
558         freeLink = header.AllocatedLinks;
559     }
560     return freeLink;
561 }
562
563 [MethodImpl(MethodImplOptions.AggressiveInlining)]
564 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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
585 public IList<TLink> GetLinkStruct(TLink linkIndex)
586 {
587     ref var link = ref GetLinkDataPartReference(linkIndex);
588     return new Link<TLink>(linkIndex, link.Source, link.Target);
589 }
590 /// <remarks>
591 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
592 ///   адрес реально поменялся
593 ///
594 /// Указатель this.links может быть в том же месте,
595 /// так как 0-я связь не используется и имеет такой же размер как Header,
596 /// поэтому header размещается в том же месте, что и 0-я связь
597 /// </remarks>
598 [MethodImpl(MethodImplOptions.AggressiveInlining)]
599 protected abstract void SetPointers(IResizableDirectMemory dataMemory,
600     IResizableDirectMemory indexMemory);
601
602 [MethodImpl(MethodImplOptions.AggressiveInlining)]
603 protected virtual void ResetPointers()
604 {
605     InternalSourcesTreeMethods = null;
606     ExternalSourcesTreeMethods = null;
607     InternalTargetsTreeMethods = null;
608     ExternalTargetsTreeMethods = null;
609     UnusedLinksListMethods = null;
610 }
611
612 [MethodImpl(MethodImplOptions.AggressiveInlining)]
613 protected abstract ref LinksHeader<TLink> GetHeaderReference();
614
615 [MethodImpl(MethodImplOptions.AggressiveInlining)]
616 protected abstract ref RawLinkDataPart<TLink> GetLinkDataPartReference(TLink linkIndex);
617
618 [MethodImpl(MethodImplOptions.AggressiveInlining)]
619 protected abstract ref RawLinkIndexPart<TLink> GetLinkIndexPartReference(TLink
620     linkIndex);
621
622 [MethodImpl(MethodImplOptions.AggressiveInlining)]
623 protected virtual bool Exists(TLink link)
624 => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
625     && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
626     && !IsUnusedLink(link);
627
628 [MethodImpl(MethodImplOptions.AggressiveInlining)]
629 protected virtual bool IsUnusedLink(TLink linkIndex)
630 {
631     if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
632         is not needed
633     {
634         // TODO: Reduce access to memory in different location (should be enough to use
635         just linkIndexPart)
636     }
637 }

```

```

631         ref var linkDataPart = ref GetLinkDataPartReference(linkIndex);
632         ref var linkIndexPart = ref GetLinkIndexPartReference(linkIndex);
633         return AreEqual(linkIndexPart.SizeAsSource, default) &&
            ↳ !AreEqual(linkDataPart.Source, default);
634     }
635     else
636     {
637         return true;
638     }
639 }
640
641 [MethodImpl(MethodImplOptions.AggressiveInlining)]
642 protected virtual TLink GetOne() => _one;
643
644 [MethodImpl(MethodImplOptions.AggressiveInlining)]
645 protected virtual TLink GetZero() => default;
646
647 [MethodImpl(MethodImplOptions.AggressiveInlining)]
648 protected virtual bool AreEqual(TLink first, TLink second) =>
    ↳ _equalityComparer.Equals(first, second);
649
650 [MethodImpl(MethodImplOptions.AggressiveInlining)]
651 protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
    ↳ second) < 0;
652
653 [MethodImpl(MethodImplOptions.AggressiveInlining)]
654 protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
    ↳ _comparer.Compare(first, second) <= 0;
655
656 [MethodImpl(MethodImplOptions.AggressiveInlining)]
657 protected virtual bool GreaterThan(TLink first, TLink second) =>
    ↳ _comparer.Compare(first, second) > 0;
658
659 [MethodImpl(MethodImplOptions.AggressiveInlining)]
660 protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
    ↳ _comparer.Compare(first, second) >= 0;
661
662 [MethodImpl(MethodImplOptions.AggressiveInlining)]
663 protected virtual long ConvertToInt64(TLink value) =>
    ↳ _addressToInt64Converter.Convert(value);
664
665 [MethodImpl(MethodImplOptions.AggressiveInlining)]
666 protected virtual TLink ConvertToAddress(long value) =>
    ↳ _int64ToAddressConverter.Convert(value);
667
668 [MethodImpl(MethodImplOptions.AggressiveInlining)]
669 protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
    ↳ second);
670
671 [MethodImpl(MethodImplOptions.AggressiveInlining)]
672 protected virtual TLink Subtract(TLink first, TLink second) =>
    ↳ Arithmetic<TLink>.Subtract(first, second);
673
674 [MethodImpl(MethodImplOptions.AggressiveInlining)]
675 protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
676
677 [MethodImpl(MethodImplOptions.AggressiveInlining)]
678 protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
679
680 #region Disposable
681
682 protected override bool AllowMultipleDisposeCalls
683 {
684     [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)]
61         protected override void SetPrevious(TLink element, TLink previous) =>
62             ↳ GetLinkDataPartReference(element).Source = previous;
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected override void SetNext(TLink element, TLink next) =>
66             ↳ GetLinkDataPartReference(element).Target = next;
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
70     }
71 }
```

1.39 ./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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public override bool Equals(object obj) => obj is RawLinkIndexPart<TLink> link ?
28             ↳ Equals(link) : false;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public bool Equals(RawLinkIndexPart<TLink> other)
32             => _equalityComparer.Equals(RootAsSource, other.RootAsSource)
33             && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
34             && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
35             && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
36             && _equalityComparer.Equals(RootAsTarget, other.RootAsTarget)
37             && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
38             && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
39             && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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/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> :
    ↪ SizedAndThreadedAVLBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
15     {
16         private static readonly uncheckedConverter<TLink, long> _addressToInt64Converter =
    ↪ uncheckedConverter<TLink, long>.Default;
17         private static readonly uncheckedConverter<TLink, int> _addressToInt32Converter =
    ↪ uncheckedConverter<TLink, int>.Default;
18         private static readonly uncheckedConverter<bool, TLink> _boolToAddressConverter =
    ↪ uncheckedConverter<bool, TLink>.Default;
19         private static readonly uncheckedConverter<TLink, bool> _addressToBoolConverter =
    ↪ uncheckedConverter<TLink, bool>.Default;
20         private static readonly uncheckedConverter<int, TLink> _int32ToAddressConverter =
    ↪ uncheckedConverter<int, TLink>.Default;
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,
    ↪ byte* header)
29         {
30             Links = links;
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
    ↪ rootSource, TLink rootTarget);
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
    ↪ rootSource, TLink rootTarget);
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
    ↪ AsRef<LinksHeader<TLink>>(Header);
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
    ↪ AsRef<RawLink<TLink>>(Links + RawLink<TLink>.SizeInBytes *
    ↪ _addressToInt64Converter.Convert(link));
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

55 protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
56 {
57     ref var link = ref GetLinkReference(linkIndex);
58     return new Link<TLink>(linkIndex, link.Source, link.Target);
59 }
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected override bool FirstIsToLeftOfSecond(TLink first, TLink second)
63 {
64     ref var firstLink = ref GetLinkReference(first);
65     ref var secondLink = ref GetLinkReference(second);
66     return FirstIsToLeftOfSecond(firstLink.Source, firstLink.Target,
        ↪ secondLink.Source, secondLink.Target);
67 }
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
71 {
72     ref var firstLink = ref GetLinkReference(first);
73     ref var secondLink = ref GetLinkReference(second);
74     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
        ↪ secondLink.Source, secondLink.Target);
75 }
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 protected virtual TLink GetSizeValue(TLink value) => Bit<TLink>.PartialRead(value, 5,
    ↪ -5);
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 protected virtual void SetSizeValue(ref TLink storedValue, TLink size) => storedValue =
    ↪ Bit<TLink>.PartialWrite(storedValue, size, 5, -5);
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 protected virtual bool GetLeftIsChildValue(TLink value)
85 {
86     unchecked
87     {
88         return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 4, 1));
89         //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 4, 1), default);
90     }
91 }
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 protected virtual void SetLeftIsChildValue(ref TLink storedValue, bool value)
95 {
96     unchecked
97     {
98         var previousValue = storedValue;
99         var modified = Bit<TLink>.PartialWrite(previousValue,
        ↪ _boolToAddressConverter.Convert(value), 4, 1);
        storedValue = modified;
100     }
101 }
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 protected virtual bool GetRightIsChildValue(TLink value)
105 {
106     unchecked
107     {
108         return _addressToBoolConverter.Convert(Bit<TLink>.PartialRead(value, 3, 1));
109         //return !EqualityComparer.Equals(Bit<TLink>.PartialRead(value, 3, 1), default);
110     }
111 }
112
113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
114 protected virtual void SetRightIsChildValue(ref TLink storedValue, bool value)
115 {
116     unchecked
117     {
118         var previousValue = storedValue;
119         var modified = Bit<TLink>.PartialWrite(previousValue,
        ↪ _boolToAddressConverter.Convert(value), 3, 1);
        storedValue = modified;
120     }
121 }
122
123 [MethodImpl(MethodImplOptions.AggressiveInlining)]
124 protected bool IsChild(TLink parent, TLink possibleChild)
125
126

```

```

127 {
128     var parentSize = GetSize(parent);
129     var childSize = GetSizeOrZero(possibleChild);
130     return GreaterThanZero(childSize) && LessOrEqualThan(childSize, parentSize);
131 }
132
133 [MethodImpl(MethodImplOptions.AggressiveInlining)]
134 protected virtual sbyte GetBalanceValue(TLink storedValue)
135 {
136     unchecked
137     {
138         var value = _addressToInt32Converter.Convert(Bit<TLink>.PartialRead(storedValue,
139             ↪ 0, 3));
140         value |= 0xF8 * ((value & 4) >> 2); // if negative, then continue ones to the
141             ↪ end of sbyte
142         return (sbyte)value;
143     }
144 }
145
146 [MethodImpl(MethodImplOptions.AggressiveInlining)]
147 protected virtual void SetBalanceValue(ref TLink storedValue, sbyte value)
148 {
149     unchecked
150     {
151         var packagedValue = _int32ToAddressConverter.Convert((byte)value >> 5 & 4 |
152             ↪ value & 3);
153         var modified = Bit<TLink>.PartialWrite(storedValue, packagedValue, 0, 3);
154         storedValue = modified;
155     }
156 }
157
158 public TLink this[TLink index]
159 {
160     [MethodImpl(MethodImplOptions.AggressiveInlining)]
161     get
162     {
163         var root = GetTreeRoot();
164         if (GreaterOrEqualThan(index, GetSize(root)))
165         {
166             return Zero;
167         }
168         while (!EqualToZero(root))
169         {
170             var left = GetLeftOrDefault(root);
171             var leftSize = GetSizeOrZero(left);
172             if (LessThan(index, leftSize))
173             {
174                 root = left;
175                 continue;
176             }
177             if (AreEqual(index, leftSize))
178             {
179                 return root;
180             }
181             root = GetRightOrDefault(root);
182             index = Subtract(index, Increment(leftSize));
183         }
184         return Zero; // TODO: Impossible situation exception (only if tree structure
185             ↪ broken)
186     }
187 }
188
189 /// <summary>
190 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
191 ↪ (концом).
192 /// </summary>
193 /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
194 /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
195 /// <returns>Индекс искомой связи.</returns>
196 [MethodImpl(MethodImplOptions.AggressiveInlining)]
197 public TLink Search(TLink source, TLink target)
198 {
199     var root = GetTreeRoot();
200     while (!EqualToZero(root))
201     {
202         ref var rootLink = ref GetLinkReference(root);
203         var rootSource = rootLink.Source;
204         var rootTarget = rootLink.Target;

```



```

200         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
201             ↳ node.Key < root.Key
202         {
203             root = GetLeftOrDefault(root);
204         }
205         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
206             ↳ node.Key > root.Key
207         {
208             root = GetRightOrDefault(root);
209         }
210         else // node.Key == root.Key
211         {
212             return root;
213         }
214     }
215     return Zero;
216 }
217
218 // TODO: Return indices range instead of references count
219 [MethodImpl(MethodImplOptions.AggressiveInlining)]
220 public TLink CountUsages(TLink link)
221 {
222     var root = GetTreeRoot();
223     var total = GetSize(root);
224     var totalRightIgnore = Zero;
225     while (!EqualToZero(root))
226     {
227         var @base = GetBasePartValue(root);
228         if (LessOrEqualThan(@base, link))
229         {
230             root = GetRightOrDefault(root);
231         }
232         else
233         {
234             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
235             root = GetLeftOrDefault(root);
236         }
237     }
238     root = GetTreeRoot();
239     var totalLeftIgnore = Zero;
240     while (!EqualToZero(root))
241     {
242         var @base = GetBasePartValue(root);
243         if (GreaterOrEqualThan(@base, link))
244         {
245             root = GetLeftOrDefault(root);
246         }
247         else
248         {
249             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
250             root = GetRightOrDefault(root);
251         }
252     }
253     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
254 }
255
256 [MethodImpl(MethodImplOptions.AggressiveInlining)]
257 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
258 {
259     var root = GetTreeRoot();
260     if (EqualToZero(root))
261     {
262         return Continue;
263     }
264     TLink first = Zero, current = root;
265     while (!EqualToZero(current))
266     {
267         var @base = GetBasePartValue(current);
268         if (GreaterOrEqualThan(@base, link))
269         {
270             if (AreEqual(@base, link))
271             {
272                 first = current;
273             }
274             current = GetLeftOrDefault(current);
275         }
276         else
277         {

```

```

277         current = GetRightOrDefault(current);
278     }
279 }
280 if (!EqualToZero(first))
281 {
282     current = first;
283     while (true)
284     {
285         if (AreEqual(handler(GetLinkValues(current)), Break))
286         {
287             return Break;
288         }
289         current = GetNext(current);
290         if (EqualToZero(current) || !AreEqual(GetBasePartValue(current), link))
291         {
292             break;
293         }
294     }
295 }
296 return Continue;
297 }
298
299 [MethodImpl(MethodImplOptions.AggressiveInlining)]
300 protected override void PrintNodeValue(TLink node, StringBuilder sb)
301 {
302     ref var link = ref GetLinkReference(node);
303     sb.Append(' ');
304     sb.Append(link.Source);
305     sb.Append('-');
306     sb.Append('>');
307     sb.Append(link.Target);
308 }
309 }
310 }

```

1.42 ./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)]

```

```

41     protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
42         ↪ rootSource, TLink rootTarget);
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
46         ↪ AsRef<LinksHeader<TLink>>(Header);
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
50         ↪ AsRef<RawLink<TLink>>(Links + RawLink<TLink>.SizeInBytes *
51         ↪ _addressToInt64Converter.Convert(link));
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
55     {
56         ref var link = ref GetLinkReference(linkIndex);
57         return new Link<TLink>(linkIndex, link.Source, link.Target);
58     }
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
62     {
63         ref var firstLink = ref GetLinkReference(first);
64         ref var secondLink = ref GetLinkReference(second);
65         return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
66             ↪ secondLink.Source, secondLink.Target);
67     }
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
71     {
72         ref var firstLink = ref GetLinkReference(first);
73         ref var secondLink = ref GetLinkReference(second);
74         return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
75             ↪ secondLink.Source, secondLink.Target);
76     }
77
78     public TLink this[TLink index]
79     {
80         [MethodImpl(MethodImplOptions.AggressiveInlining)]
81         get
82         {
83             var root = GetTreeRoot();
84             if (GreaterOrEqualThan(index, GetSize(root)))
85             {
86                 return Zero;
87             }
88             while (!EqualToZero(root))
89             {
90                 var left = GetLeftOrDefault(root);
91                 var leftSize = GetSizeOrZero(left);
92                 if (LessThan(index, leftSize))
93                 {
94                     root = left;
95                     continue;
96                 }
97                 if (AreEqual(index, leftSize))
98                 {
99                     return root;
100                 }
101                 root = GetRightOrDefault(root);
102                 index = Subtract(index, Increment(leftSize));
103             }
104             return Zero; // TODO: Impossible situation exception (only if tree structure
105                 ↪ broken)
106         }
107     }
108
109     /// <summary>
110     /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
111     /// ↪ (концом).
112     /// </summary>
113     /// <param name="source">Индекс связи, которая является началом на искомой связи.</param>
114     /// <param name="target">Индекс связи, которая является концом на искомой связи.</param>
115     /// <returns>Индекс искомой связи.</returns>
116     [MethodImpl(MethodImplOptions.AggressiveInlining)]
117     public TLink Search(TLink source, TLink target)
118     {

```

```

111     var root = GetTreeRoot();
112     while (!EqualToZero(root))
113     {
114         ref var rootLink = ref GetLinkReference(root);
115         var rootSource = rootLink.Source;
116         var rootTarget = rootLink.Target;
117         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
118             ↪ node.Key < root.Key
119         {
120             root = GetLeftOrDefault(root);
121         }
122         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget)) //
123             ↪ node.Key > root.Key
124         {
125             root = GetRightOrDefault(root);
126         }
127         else // node.Key == root.Key
128         {
129             return root;
130         }
131     }
132     return Zero;
133 }
134
135 // TODO: Return indices range instead of references count
136 [MethodImpl(MethodImplOptions.AggressiveInlining)]
137 public TLink CountUsages(TLink link)
138 {
139     var root = GetTreeRoot();
140     var total = GetSize(root);
141     var totalRightIgnore = Zero;
142     while (!EqualToZero(root))
143     {
144         var @base = GetBasePartValue(root);
145         if (LessOrEqualThan(@base, link))
146         {
147             root = GetRightOrDefault(root);
148         }
149         else
150         {
151             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
152             root = GetLeftOrDefault(root);
153         }
154     }
155     root = GetTreeRoot();
156     var totalLeftIgnore = Zero;
157     while (!EqualToZero(root))
158     {
159         var @base = GetBasePartValue(root);
160         if (GreaterOrEqualThan(@base, link))
161         {
162             root = GetLeftOrDefault(root);
163         }
164         else
165         {
166             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
167             root = GetRightOrDefault(root);
168         }
169     }
170     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
171 }
172
173 [MethodImpl(MethodImplOptions.AggressiveInlining)]
174 public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
175     ↪ EachUsageCore(@base, GetTreeRoot(), handler);
176
177 // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
178 ↪ low-level MSIL stack.
179 [MethodImpl(MethodImplOptions.AggressiveInlining)]
180 private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
181 {
182     var @continue = Continue;
183     if (EqualToZero(link))
184     {
185         return @continue;
186     }
187     var linkBasePart = GetBasePartValue(link);
188     var @break = Break;
189     if (GreaterThan(linkBasePart, @base))

```

```

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

```

1.43 ./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 }

```

```

29
30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 protected override TLink GetSize(TLink node) =>
    ↳ GetSizeValue(GetLinkReference(node).SizeAsSource);
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
    ↳ GetLinkReference(node).SizeAsSource, size);
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 protected override bool GetLeftIsChild(TLink node) =>
    ↳ GetLeftIsChildValue(GetLinkReference(node).SizeAsSource);
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 protected override void SetLeftIsChild(TLink node, bool value) =>
    ↳ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected override bool GetRightIsChild(TLink node) =>
    ↳ GetRightIsChildValue(GetLinkReference(node).SizeAsSource);
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 protected override void SetRightIsChild(TLink node, bool value) =>
    ↳ SetRightIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 protected override sbyte GetBalance(TLink node) =>
    ↳ GetBalanceValue(GetLinkReference(node).SizeAsSource);
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
    ↳ GetLinkReference(node).SizeAsSource, value);
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected override TLink GetTreeRoot() => GetHeaderReference().RootAsSource;
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
    ↳ 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 }

```

1.44 ./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,
    ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
    ↪ AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget);
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     protected override void ClearNode(TLink node)
50     {
51         ref var link = ref GetLinkReference(node);
52         link.LeftAsSource = Zero;
53         link.RightAsSource = Zero;
54         link.SizeAsSource = Zero;
55     }
56 }
57 }

```

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

```

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

```

```

23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     protected override void SetLeft(TLink node, TLink left) =>
25     ↪ GetLinkReference(node).LeftAsTarget = left;
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     protected override void SetRight(TLink node, TLink right) =>
29     ↪ GetLinkReference(node).RightAsTarget = right;
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected override TLink GetSize(TLink node) =>
33     ↪ GetSizeValue(GetLinkReference(node).SizeAsTarget);
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
37     ↪ GetLinkReference(node).SizeAsTarget, size);
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected override bool GetLeftIsChild(TLink node) =>
41     ↪ GetLeftIsChildValue(GetLinkReference(node).SizeAsTarget);
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override void SetLeftIsChild(TLink node, bool value) =>
45     ↪ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override bool GetRightIsChild(TLink node) =>
49     ↪ GetRightIsChildValue(GetLinkReference(node).SizeAsTarget);
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override void SetRightIsChild(TLink node, bool value) =>
53     ↪ SetRightIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected override sbyte GetBalance(TLink node) =>
57     ↪ GetBalanceValue(GetLinkReference(node).SizeAsTarget);
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
61     ↪ GetLinkReference(node).SizeAsTarget, value);
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected override TLink GetTreeRoot() => GetHeaderReference().RootAsTarget;
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
71     ↪ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
72     ↪ AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource);
73
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
76     ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
77     ↪ AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource);
78
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     protected override void ClearNode(TLink node)
81     {
82         ref var link = ref GetLinkReference(node);
83         link.LeftAsTarget = Zero;
84         link.RightAsTarget = Zero;
85         link.SizeAsTarget = Zero;
86     }
87 }

```

1.46 ./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     {

```



```

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

```

1.47 ./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     {
56         _createSourceTreeMethods = () => new
57             ↳ LinksSourcesSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
58         _createTargetTreeMethods = () => new
59             ↳ LinksTargetsSizeBalancedTreeMethods<TLink>(Constants, _links, _header);
60     }
61     Init(memory, memoryReservationStep);
62 }
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 protected override void SetPointers(IResizableDirectMemory memory)
66 {
67     _links = (byte*)memory.Pointer;
68     _header = _links;
69     SourcesTreeMethods = _createSourceTreeMethods();
70     TargetsTreeMethods = _createTargetTreeMethods();
71     UnusedLinksListMethods = new UnusedLinksListMethods<TLink>(_links, _header);
72 }
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected override void ResetPointers()
76 {
77     base.ResetPointers();
78     _links = null;
79     _header = null;
80 }
81
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 protected override ref LinksHeader<TLink> GetHeaderReference() => ref
84   ↳ AsRef<LinksHeader<TLink>>(_header);
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 protected override ref RawLink<TLink> GetLinkReference(TLink linkIndex) => ref
88   ↳ AsRef<RawLink<TLink>>(_links + LinkSizeInBytes * ConvertToInt64(linkIndex));
89 }
90 }

```

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

```

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

```

```

5 using Platform.Singletons;
6 using Platform.Converters;
7 using Platform.Numbers;
8 using Platform.Memory;
9 using Platform.Data.Exceptions;
10
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Data.Doublets.Memory.United.Generic
14 {
15     public abstract class UnitedMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16     {
17         private static readonly EqualityComparer<TLink> _equalityComparer =
18             ↪ EqualityComparer<TLink>.Default;
19         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
20         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
21             ↪ UncheckedConverter<TLink, long>.Default;
22         private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
23             ↪ UncheckedConverter<long, TLink>.Default;
24
25         private static readonly TLink _zero = default;
26         private static readonly TLink _one = Arithmetic.Increment(_zero);
27
28         /// <summary>Возвращает размер одной связи в байтах.</summary>
29         /// <remarks>
30         /// Используется только во вне класса, не рекомендуется использовать внутри.
31         /// Так как во вне не обязательно будет доступен unsafe C#.
32         /// </remarks>
33         public static readonly long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;
34
35         public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
36
37         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
38
39         protected readonly IResizableDirectMemory _memory;
40         protected readonly long _memoryReservationStep;
41
42         protected ILinksTreeMethods<TLink> TargetsTreeMethods;
43         protected ILinksTreeMethods<TLink> SourcesTreeMethods;
44         // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
45         ↪ нужно использовать не список а дерево, так как так можно быстрее проверить на
46         ↪ наличие связи внутри
47         protected ILinksListMethods<TLink> UnusedLinksListMethods;
48
49         /// <summary>
50         /// Возвращает общее число связей находящихся в хранилище.
51         /// </summary>
52         protected virtual TLink Total
53         {
54             [MethodImpl(MethodImplOptions.AggressiveInlining)]
55             get
56             {
57                 ref var header = ref GetHeaderReference();
58                 return Subtract(header.AllocatedLinks, header.FreeLinks);
59             }
60         }
61
62         public virtual LinksConstants<TLink> Constants
63         {
64             [MethodImpl(MethodImplOptions.AggressiveInlining)]
65             get;
66         }
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
70             ↪ memoryReservationStep, LinksConstants<TLink> constants)
71         {
72             _memory = memory;
73             _memoryReservationStep = memoryReservationStep;
74             Constants = constants;
75         }
76
77         [MethodImpl(MethodImplOptions.AggressiveInlining)]
78         protected UnitedMemoryLinksBase(IResizableDirectMemory memory, long
79             ↪ memoryReservationStep) : this(memory, memoryReservationStep,
80             ↪ Default<LinksConstants<TLink>>.Instance) { }
81
82         [MethodImpl(MethodImplOptions.AggressiveInlining)]
83         protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
84         {
85             if (memory.ReservedCapacity < memoryReservationStep)

```

```

78     {
79         memory.ReservedCapacity = memoryReservationStep;
80     }
81     SetPointers(memory);
82     ref var header = ref GetHeaderReference();
83     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
84     memory.UsedCapacity = ConvertToInt64(header.AllocatedLinks) * LinkSizeInBytes +
85         ↳ LinkHeaderSizeInBytes;
86     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
87     header.ReservedLinks = ConvertToAddress((memory.ReservedCapacity -
88         ↳ LinkHeaderSizeInBytes) / LinkSizeInBytes);
89 }
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 public virtual TLink Count(IList<TLink> restrictions)
92 {
93     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
94     if (restrictions.Count == 0)
95     {
96         return Total;
97     }
98     var constants = Constants;
99     var any = constants.Any;
100    var index = restrictions[constants.IndexPart];
101    if (restrictions.Count == 1)
102    {
103        if (AreEqual(index, any))
104        {
105            return Total;
106        }
107        return Exists(index) ? GetOne() : GetZero();
108    }
109    if (restrictions.Count == 2)
110    {
111        var value = restrictions[1];
112        if (AreEqual(index, any))
113        {
114            if (AreEqual(value, any))
115            {
116                return Total; // Any - как отсутствие ограничения
117            }
118            return Add(SourcesTreeMethods.CountUsages(value),
119                ↳ TargetsTreeMethods.CountUsages(value));
120        }
121        else
122        {
123            if (!Exists(index))
124            {
125                return GetZero();
126            }
127            if (AreEqual(value, any))
128            {
129                return GetOne();
130            }
131            ref var storedLinkValue = ref GetLinkReference(index);
132            if (AreEqual(storedLinkValue.Source, value) ||
133                ↳ AreEqual(storedLinkValue.Target, value))
134            {
135                return GetOne();
136            }
137            return GetZero();
138        }
139    }
140    if (restrictions.Count == 3)
141    {
142        var source = restrictions[constants.SourcePart];
143        var target = restrictions[constants.TargetPart];
144        if (AreEqual(index, any))
145        {
146            if (AreEqual(source, any) && AreEqual(target, any))
147            {
148                return Total;
149            }
150            else if (AreEqual(source, any))
151            {
152                return TargetsTreeMethods.CountUsages(target);
153            }
154            else if (AreEqual(target, any))

```

```

152     {
153         return SourcesTreeMethods.CountUsages(source);
154     }
155     else //if(source != Any && target != Any)
156     {
157         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
158         var link = SourcesTreeMethods.Search(source, target);
159         return AreEqual(link, constants.Null) ? GetZero() : GetOne();
160     }
161 }
162 else
163 {
164     if (!Exists(index))
165     {
166         return GetZero();
167     }
168     if (AreEqual(source, any) && AreEqual(target, any))
169     {
170         return GetOne();
171     }
172     ref var storedLinkValue = ref GetLinkReference(index);
173     if (!AreEqual(source, any) && !AreEqual(target, any))
174     {
175         if (AreEqual(storedLinkValue.Source, source) &&
176             ↪ AreEqual(storedLinkValue.Target, target))
177         {
178             return GetOne();
179         }
180         return GetZero();
181     }
182     var value = default(TLink);
183     if (AreEqual(source, any))
184     {
185         value = target;
186     }
187     if (AreEqual(target, any))
188     {
189         value = source;
190     }
191     if (AreEqual(storedLinkValue.Source, value) ||
192         ↪ AreEqual(storedLinkValue.Target, value))
193     {
194         return GetOne();
195     }
196     return GetZero();
197 }
198 }
199
200 [MethodImpl(MethodImplOptions.AggressiveInlining)]
201 public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
202 {
203     var constants = Constants;
204     var @break = constants.Break;
205     if (restrictions.Count == 0)
206     {
207         for (var link = GetOne(); LessOrEqualThan(link,
208             ↪ GetHeaderReference().AllocatedLinks); link = Increment(link))
209         {
210             if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
211             {
212                 return @break;
213             }
214         }
215         return @break;
216     }
217     var @continue = constants.Continue;
218     var any = constants.Any;
219     var index = restrictions[constants.IndexPart];
220     if (restrictions.Count == 1)
221     {
222         if (AreEqual(index, any))
223         {
224             return Each(handler, Array.Empty<TLink>());
225         }
226         if (!Exists(index))

```

```

226     {
227         return @continue;
228     }
229     return handler(GetLinkStruct(index));
230 }
231 if (restrictions.Count == 2)
232 {
233     var value = restrictions[1];
234     if (AreEqual(index, any))
235     {
236         if (AreEqual(value, any))
237         {
238             return Each(handler, Array.Empty<TLink>());
239         }
240         if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
241         {
242             return @break;
243         }
244         return Each(handler, new Link<TLink>(index, any, value));
245     }
246     else
247     {
248         if (!Exists(index))
249         {
250             return @continue;
251         }
252         if (AreEqual(value, any))
253         {
254             return handler(GetLinkStruct(index));
255         }
256         ref var storedLinkValue = ref GetLinkReference(index);
257         if (AreEqual(storedLinkValue.Source, value) ||
258             AreEqual(storedLinkValue.Target, value))
259         {
260             return handler(GetLinkStruct(index));
261         }
262         return @continue;
263     }
264 }
265 if (restrictions.Count == 3)
266 {
267     var source = restrictions[constants.SourcePart];
268     var target = restrictions[constants.TargetPart];
269     if (AreEqual(index, any))
270     {
271         if (AreEqual(source, any) && AreEqual(target, any))
272         {
273             return Each(handler, Array.Empty<TLink>());
274         }
275         else if (AreEqual(source, any))
276         {
277             return TargetsTreeMethods.EachUsage(target, handler);
278         }
279         else if (AreEqual(target, any))
280         {
281             return SourcesTreeMethods.EachUsage(source, handler);
282         }
283         else //if(source != Any && target != Any)
284         {
285             var link = SourcesTreeMethods.Search(source, target);
286             return AreEqual(link, constants.Null) ? @continue :
287                 ↪ handler(GetLinkStruct(link));
288         }
289     }
290     else
291     {
292         if (!Exists(index))
293         {
294             return @continue;
295         }
296         if (AreEqual(source, any) && AreEqual(target, any))
297         {
298             return handler(GetLinkStruct(index));
299         }
300         ref var storedLinkValue = ref GetLinkReference(index);
301         if (!AreEqual(source, any) && !AreEqual(target, any))
302         {
303             if (AreEqual(storedLinkValue.Source, source) &&

```

```

303         AreEqual(storedLinkValue.Target, target))
304     {
305         return handler(GetLinkStruct(index));
306     }
307     return @continue;
308 }
309 var value = default(TLink);
310 if (AreEqual(source, any))
311 {
312     value = target;
313 }
314 if (AreEqual(target, any))
315 {
316     value = source;
317 }
318 if (AreEqual(storedLinkValue.Source, value) ||
319     AreEqual(storedLinkValue.Target, value))
320 {
321     return handler(GetLinkStruct(index));
322 }
323 return @continue;
324 }
325 }
326 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
327 }
328
329 /// <remarks>
330 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↳ в другом месте (но не в менеджере памяти, а в логике Links)
331 /// </remarks>
332 [MethodImpl(MethodImplOptions.AggressiveInlining)]
333 public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
334 {
335     var constants = Constants;
336     var @null = constants.Null;
337     var linkIndex = restrictions[constants.IndexPart];
338     ref var link = ref GetLinkReference(linkIndex);
339     ref var header = ref GetHeaderReference();
340     ref var firstAsSource = ref header.RootAsSource;
341     ref var firstAsTarget = ref header.RootAsTarget;
342     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↳ предварительно заполнено нулями
343     if (!AreEqual(link.Source, @null))
344     {
345         SourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
346     }
347     if (!AreEqual(link.Target, @null))
348     {
349         TargetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
350     }
351     link.Source = substitution[constants.SourcePart];
352     link.Target = substitution[constants.TargetPart];
353     if (!AreEqual(link.Source, @null))
354     {
355         SourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
356     }
357     if (!AreEqual(link.Target, @null))
358     {
359         TargetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
360     }
361     return linkIndex;
362 }
363
364 /// <remarks>
365 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
    ↳ пространство
366 /// </remarks>
367 [MethodImpl(MethodImplOptions.AggressiveInlining)]
368 public virtual TLink Create(IList<TLink> restrictions)
369 {
370     ref var header = ref GetHeaderReference();
371     var freeLink = header.FirstFreeLink;
372     if (!AreEqual(freeLink, Constants.Null))
373     {
374         UnusedLinksListMethods.Detach(freeLink);
375     }
376     else
377     {

```

```

378     var maximumPossibleInnerReference = Constants.InternalReferencesRange.Maximum;
379     if (GreaterThan(header.AllocatedLinks, maximumPossibleInnerReference))
380     {
381         throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
382     }
383     if (GreaterOrEqualThan(header.AllocatedLinks, Decrement(header.ReservedLinks)))
384     {
385         _memory.ReservedCapacity += _memory.ReservationStep;
386         SetPointers(_memory);
387         header = ref GetHeaderReference();
388         header.ReservedLinks = ConvertToAddress(_memory.ReservedCapacity /
389             ↳ LinkSizeInBytes);
390     }
391     header.AllocatedLinks = Increment(header.AllocatedLinks);
392     _memory.UsedCapacity += LinkSizeInBytes;
393     freeLink = header.AllocatedLinks;
394 }
395
396 [MethodImpl(MethodImplOptions.AggressiveInlining)]
397 public virtual void Delete(IList<TLink> restrictions)
398 {
399     ref var header = ref GetHeaderReference();
400     var link = restrictions[Constants.IndexPart];
401     if (LessThan(link, header.AllocatedLinks))
402     {
403     }
404     UnusedLinksListMethods.AttachAsFirst(link);
405 }
406 else if (AreEqual(link, header.AllocatedLinks))
407 {
408     header.AllocatedLinks = Decrement(header.AllocatedLinks);
409     _memory.UsedCapacity -= LinkSizeInBytes;
410     // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
411     ↳ пока не дойдём до первой существующей связи
412     // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
413     while (GreaterThan(header.AllocatedLinks, GetZero()) &&
414         ↳ IsUnusedLink(header.AllocatedLinks))
415     {
416         UnusedLinksListMethods.Detach(header.AllocatedLinks);
417         header.AllocatedLinks = Decrement(header.AllocatedLinks);
418         _memory.UsedCapacity -= LinkSizeInBytes;
419     }
420 }
421 }
422
423 [MethodImpl(MethodImplOptions.AggressiveInlining)]
424 public IList<TLink> GetLinkStruct(TLink linkIndex)
425 {
426     ref var link = ref GetLinkReference(linkIndex);
427     return new Link<TLink>(linkIndex, link.Source, link.Target);
428 }
429
430 /// <remarks>
431 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
432 ↳ адрес реально поменялся
433 ///
434 /// Указатель this.links может быть в том же месте,
435 /// так как 0-я связь не используется и имеет такой же размер как Header,
436 /// поэтому header размещается в том же месте, что и 0-я связь
437 /// </remarks>
438 [MethodImpl(MethodImplOptions.AggressiveInlining)]
439 protected abstract void SetPointers(IResizableDirectMemory memory);
440
441 [MethodImpl(MethodImplOptions.AggressiveInlining)]
442 protected virtual void ResetPointers()
443 {
444     SourcesTreeMethods = null;
445     TargetsTreeMethods = null;
446     UnusedLinksListMethods = null;
447 }
448
449 [MethodImpl(MethodImplOptions.AggressiveInlining)]
450 protected abstract ref LinksHeader<TLink> GetHeaderReference();
451
452 [MethodImpl(MethodImplOptions.AggressiveInlining)]
453 protected abstract ref RawLink<TLink> GetLinkReference(TLink linkIndex);
454
455 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

453     protected virtual bool Exists(TLink link)
454         => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
455         && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
456         && !IsUnusedLink(link);
457
458     [MethodImpl(MethodImplOptions.AggressiveInlining)]
459     protected virtual bool IsUnusedLink(TLink linkIndex)
460     {
461         if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
462             ↳ is not needed
463         {
464             ref var link = ref GetLinkReference(linkIndex);
465             return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
466         }
467         else
468         {
469             return true;
470         }
471     }
472
473     [MethodImpl(MethodImplOptions.AggressiveInlining)]
474     protected virtual TLink GetOne() => _one;
475
476     [MethodImpl(MethodImplOptions.AggressiveInlining)]
477     protected virtual TLink GetZero() => default;
478
479     [MethodImpl(MethodImplOptions.AggressiveInlining)]
480     protected virtual bool AreEqual(TLink first, TLink second) =>
481         ↳ _equalityComparer.Equals(first, second);
482
483     [MethodImpl(MethodImplOptions.AggressiveInlining)]
484     protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
485         ↳ second) < 0;
486
487     [MethodImpl(MethodImplOptions.AggressiveInlining)]
488     protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
489         ↳ _comparer.Compare(first, second) <= 0;
490
491     [MethodImpl(MethodImplOptions.AggressiveInlining)]
492     protected virtual bool GreaterThan(TLink first, TLink second) =>
493         ↳ _comparer.Compare(first, second) > 0;
494
495     [MethodImpl(MethodImplOptions.AggressiveInlining)]
496     protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
497         ↳ _comparer.Compare(first, second) >= 0;
498
499     [MethodImpl(MethodImplOptions.AggressiveInlining)]
500     protected virtual long ConvertToInt64(TLink value) =>
501         ↳ _addressToInt64Converter.Convert(value);
502
503     [MethodImpl(MethodImplOptions.AggressiveInlining)]
504     protected virtual TLink ConvertToAddress(long value) =>
505         ↳ _int64ToAddressConverter.Convert(value);
506
507     [MethodImpl(MethodImplOptions.AggressiveInlining)]
508     protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
509         ↳ second);
510
511     [MethodImpl(MethodImplOptions.AggressiveInlining)]
512     protected virtual TLink Subtract(TLink first, TLink second) =>
513         ↳ Arithmetic<TLink>.Subtract(first, second);
514
515     [MethodImpl(MethodImplOptions.AggressiveInlining)]
516     protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
517
518     [MethodImpl(MethodImplOptions.AggressiveInlining)]
519     protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
520
521     #region Disposable
522
523     protected override bool AllowMultipleDisposeCalls
524     {
525         [MethodImpl(MethodImplOptions.AggressiveInlining)]
526         get => true;
527     }
528
529     [MethodImpl(MethodImplOptions.AggressiveInlining)]
530     protected override void Dispose(bool manual, bool wasDisposed)
531     {

```

```

522         if (!wasDisposed)
523         {
524             ResetPointers();
525             _memory.DisposeIfPossible();
526         }
527     }
528
529     #endregion
530 }
531 }

```

1.49 ./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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
28             ↳ AsRef<LinksHeader<TLink>>(_header);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
32             ↳ AsRef<RawLink<TLink>>(_links + RawLink<TLink>.SizeInBytes *
33             ↳ _addressToInt64Converter.Convert(link));
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override TLink GetLast() => GetHeaderReference().LastFreeLink;
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected override TLink GetPrevious(TLink element) => GetLinkReference(element).Source;
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected override TLink GetNext(TLink element) => GetLinkReference(element).Target;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected override TLink GetSize() => GetHeaderReference().FreeLinks;
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
52             ↳ element;
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
56             ↳ element;
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         protected override void SetPrevious(TLink element, TLink previous) =>
60             ↳ GetLinkReference(element).Source = previous;
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         protected override void SetNext(TLink element, TLink next) =>
64             ↳ GetLinkReference(element).Target = next;
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
68     }
69 }

```

```
60 }
```

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

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

1.51 ./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 }
```

```

21 [MethodImpl(MethodImplOptions.AggressiveInlining)]
22 protected override ulong GetZero() => 0UL;
23
24 [MethodImpl(MethodImplOptions.AggressiveInlining)]
25 protected override bool EqualToZero(ulong value) => value == 0UL;
26
27 [MethodImpl(MethodImplOptions.AggressiveInlining)]
28 protected override bool AreEqual(ulong first, ulong second) => first == second;
29
30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 protected override bool GreaterThanZero(ulong value) => value > 0UL;
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 protected override bool GreaterThan(ulong first, ulong second) => first > second;
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
38
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
    ↳ always true for ulong
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
    ↳ always >= 0 for ulong
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
    ↳ for ulong
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 protected override bool LessThan(ulong first, ulong second) => first < second;
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected override ulong Increment(ulong value) => ++value;
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 protected override ulong Decrement(ulong value) => --value;
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected override ulong Add(ulong first, ulong second) => first + second;
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 protected override ulong Subtract(ulong first, ulong second) => first - second;
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
68 {
69     ref var firstLink = ref Links[first];
70     ref var secondLink = ref Links[second];
71     return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
    ↳ secondLink.Source, secondLink.Target);
72 }
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
76 {
77     ref var firstLink = ref Links[first];
78     ref var secondLink = ref Links[second];
79     return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
    ↳ secondLink.Source, secondLink.Target);
80 }
81
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 protected override ulong GetSizeValue(ulong value) => (value & 4294967264UL) >> 5;
84
85 [MethodImpl(MethodImplOptions.AggressiveInlining)]
86 protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
    ↳ storedValue & 31UL | (size & 134217727UL) << 5;
87
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 protected override bool GetLeftIsChildValue(ulong value) => (value & 16UL) >> 4 == 1UL;
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
    ↳ storedValue = storedValue & 4294967279UL | (As<bool, byte>(ref value) & 1UL) << 4;

```

```

93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     protected override bool GetRightIsChildValue(ulong value) => (value & 8UL) >> 3 == 1UL;
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>
98     ↪ storedValue = storedValue & 4294967287UL | (As<bool, byte>(ref value) & 1UL) << 3;
99
100    [MethodImpl(MethodImplOptions.AggressiveInlining)]
101    protected override sbyte GetBalanceValue(ulong value) => unchecked((sbyte)(value & 7UL |
102    ↪ 0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of
103    ↪ sbyte
104
105    [MethodImpl(MethodImplOptions.AggressiveInlining)]
106    protected override void SetBalanceValue(ref ulong storedValue, sbyte value) =>
107    ↪ storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
108    ↪ value & 3) & 7UL);
109
110    [MethodImpl(MethodImplOptions.AggressiveInlining)]
111    protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
112
113    [MethodImpl(MethodImplOptions.AggressiveInlining)]
114    protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
115 }
116 }

```

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

```

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

```

```

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

```

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

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Memory.United.Specific
6  {
7      public unsafe class UInt64LinksSourcesAvlBalancedTreeMethods :
8          ↪ UInt64LinksAvlBalancedTreeMethodsBase
9      {
10         public UInt64LinksSourcesAvlBalancedTreeMethods(LinksConstants<ulong> constants,
11             ↪ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
12             ↪ { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetLeftReference(ulong node) => ref
16             ↪ Links[node].LeftAsSource;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20             ↪ Links[node].RightAsSource;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
30             ↪ left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
34             ↪ right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

30     protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsSource);
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
34         ↳ Links[node].SizeAsSource, size);
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected override bool GetLeftIsChild(ulong node) =>
38         ↳ GetLeftIsChildValue(Links[node].SizeAsSource);
39
40     //[MethodImpl(MethodImplOptions.AggressiveInlining)]
41     //protected override bool GetLeftIsChild(ulong node) => IsChild(node, GetLeft(node));
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     protected override void SetLeftIsChild(ulong node, bool value) =>
45         ↳ SetLeftIsChildValue(ref Links[node].SizeAsSource, value);
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override bool GetRightIsChild(ulong node) =>
49         ↳ GetRightIsChildValue(Links[node].SizeAsSource);
50
51     //[MethodImpl(MethodImplOptions.AggressiveInlining)]
52     //protected override bool GetRightIsChild(ulong node) => IsChild(node, GetRight(node));
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected override void SetRightIsChild(ulong node, bool value) =>
56         ↳ SetRightIsChildValue(ref Links[node].SizeAsSource, value);
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     protected override sbyte GetBalance(ulong node) =>
60         ↳ GetBalanceValue(Links[node].SizeAsSource);
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
64         ↳ Links[node].SizeAsSource, value);
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected override ulong GetTreeRoot() => Header->RootAsSource;
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
74         ↳ ulong secondSource, ulong secondTarget)
75         => firstSource < secondSource || firstSource == secondSource && firstTarget <
76         ↳ secondTarget;
77
78     [MethodImpl(MethodImplOptions.AggressiveInlining)]
79     protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
80         ↳ ulong secondSource, ulong secondTarget)
81         => firstSource > secondSource || firstSource == secondSource && firstTarget >
82         ↳ secondTarget;
83
84     [MethodImpl(MethodImplOptions.AggressiveInlining)]
85     protected override void ClearNode(ulong node)
86     {
87         ref var link = ref Links[node];
88         link.LeftAsSource = OUL;
89         link.RightAsSource = OUL;
90         link.SizeAsSource = OUL;
91     }
92 }

```

1.54 ./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 }

```

```

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

```

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

```

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

```



```

16 [MethodImpl(MethodImplOptions.AggressiveInlining)]
17 protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
18
19 [MethodImpl(MethodImplOptions.AggressiveInlining)]
20 protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
21
22 [MethodImpl(MethodImplOptions.AggressiveInlining)]
23 protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
24     ↳ left;
25
26 [MethodImpl(MethodImplOptions.AggressiveInlining)]
27 protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
28     ↳ right;
29
30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsTarget);
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
35     ↳ Links[node].SizeAsTarget, size);
36
37 [MethodImpl(MethodImplOptions.AggressiveInlining)]
38 protected override bool GetLeftIsChild(ulong node) =>
39     ↳ GetLeftIsChildValue(Links[node].SizeAsTarget);
40
41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 protected override void SetLeftIsChild(ulong node, bool value) =>
43     ↳ SetLeftIsChildValue(ref Links[node].SizeAsTarget, value);
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 protected override bool GetRightIsChild(ulong node) =>
47     ↳ GetRightIsChildValue(Links[node].SizeAsTarget);
48
49 [MethodImpl(MethodImplOptions.AggressiveInlining)]
50 protected override void SetRightIsChild(ulong node, bool value) =>
51     ↳ SetRightIsChildValue(ref Links[node].SizeAsTarget, value);
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 protected override sbyte GetBalance(ulong node) =>
55     ↳ GetBalanceValue(Links[node].SizeAsTarget);
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
59     ↳ Links[node].SizeAsTarget, value);
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected override ulong GetTreeRoot() => Header->RootAsTarget;
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
69     ↳ ulong secondSource, ulong secondTarget)
70     => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
71     ↳ secondSource;
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
75     ↳ ulong secondSource, ulong secondTarget)
76     => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
77     ↳ secondSource;
78
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 protected override void ClearNode(ulong node)
81 {
82     ref var link = ref Links[node];
83     link.LeftAsTarget = OUL;
84     link.RightAsTarget = OUL;
85     link.SizeAsTarget = OUL;
86 }
87 }
88 }

```

```

1.56 ./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs
1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

```

```

4
5 namespace Platform.Data.Doublets.Memory.United.Specific
6 {
7     public unsafe class UInt64LinksTargetsSizeBalancedTreeMethods :
8         ↳ UInt64LinksSizeBalancedTreeMethodsBase
9     {
10         public UInt64LinksTargetsSizeBalancedTreeMethods(LinksConstants<ulong> constants,
11             ↳ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
12             ↳ { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override ref ulong GetLeftReference(ulong node) => ref
16             ↳ Links[node].LeftAsTarget;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override ref ulong GetRightReference(ulong node) => ref
20             ↳ Links[node].RightAsTarget;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
30             ↳ left;
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
34             ↳ right;
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override void SetSize(ulong node, ulong size) => Links[node].SizeAsTarget =
41             ↳ size;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected override ulong GetTreeRoot() => Header->RootAsTarget;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
51             ↳ ulong secondSource, ulong secondTarget)
52             ↳ => firstTarget < secondTarget || firstTarget == secondTarget && firstSource <
53                 ↳ secondSource;
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
57             ↳ ulong secondSource, ulong secondTarget)
58             ↳ => firstTarget > secondTarget || firstTarget == secondTarget && firstSource >
59                 ↳ secondSource;
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         protected override void ClearNode(ulong node)
63         {
64             ref var link = ref Links[node];
65             link.LeftAsTarget = OUL;
66             link.RightAsTarget = OUL;
67             link.SizeAsTarget = OUL;
68         }
69     }
70 }

```

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

```

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

```

```

11  /// <summary>
12  /// <para>Represents a low-level implementation of direct access to resizable memory, for
    ↳ organizing the storage of links with addresses represented as <see cref="ulong"
    ↳ />.</para>
13  /// <para>Представляет низкоуровневую реализация прямого доступа к памяти с переменным
    ↳ размером, для организации хранения связей с адресами представленными в виде <see
    ↳ cref="ulong"/>.</para>
14  /// </summary>
15  public unsafe class UInt64UnitedMemoryLinks : UnitedMemoryLinksBase<ulong>
16  {
17      private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
18      private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
19      private LinksHeader<ulong>* _header;
20      private RawLink<ulong>* _links;
21
22      [MethodImpl(MethodImplOptions.AggressiveInlining)]
23      public UInt64UnitedMemoryLinks(string address) : this(address, DefaultLinksSizeStep) { }
24
25      /// <summary>
26      /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
    ↳ минимальным шагом расширения базы данных.
27      /// </summary>
28      /// <param name="address">Полный путь к файлу базы данных.</param>
29      /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
    ↳ байтах.</param>
30      [MethodImpl(MethodImplOptions.AggressiveInlining)]
31      public UInt64UnitedMemoryLinks(string address, long memoryReservationStep) : this(new
    ↳ FileMappedResizableDirectMemory(address, memoryReservationStep),
    ↳ memoryReservationStep) { }
32
33      [MethodImpl(MethodImplOptions.AggressiveInlining)]
34      public UInt64UnitedMemoryLinks(IResizableDirectMemory memory) : this(memory,
    ↳ DefaultLinksSizeStep) { }
35
36      [MethodImpl(MethodImplOptions.AggressiveInlining)]
37      public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
    ↳ memoryReservationStep) : this(memory, memoryReservationStep,
    ↳ Default<LinksConstants<ulong>>.Instance, IndexTreeType.Default) { }
38
39      [MethodImpl(MethodImplOptions.AggressiveInlining)]
40      public UInt64UnitedMemoryLinks(IResizableDirectMemory memory, long
    ↳ memoryReservationStep, LinksConstants<ulong> constants, IndexTreeType indexTreeType)
    ↳ : base(memory, memoryReservationStep, constants)
41      {
42          if (indexTreeType == IndexTreeType.SizedAndThreadedAVLBalancedTree)
43          {
44              _createSourceTreeMethods = () => new
    ↳ UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
45              _createTargetTreeMethods = () => new
    ↳ UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
46          }
47          else
48          {
49              _createSourceTreeMethods = () => new
    ↳ UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
50              _createTargetTreeMethods = () => new
    ↳ UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
51          }
52          Init(memory, memoryReservationStep);
53      }
54
55      [MethodImpl(MethodImplOptions.AggressiveInlining)]
56      protected override void SetPointers(IResizableDirectMemory memory)
57      {
58          _header = (LinksHeader<ulong>*)memory.Pointer;
59          _links = (RawLink<ulong>*)memory.Pointer;
60          SourcesTreeMethods = _createSourceTreeMethods();
61          TargetsTreeMethods = _createTargetTreeMethods();
62          UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
63      }
64
65      [MethodImpl(MethodImplOptions.AggressiveInlining)]
66      protected override void ResetPointers()
67      {
68          base.ResetPointers();
69          _links = null;
70          _header = null;
71      }

```

```

72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
74
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
77         ↪ _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 }

```

1.58 ./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 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Reflection;
3  using Platform.Converters;
4  using Platform.Numbers;
5  using System.Runtime.CompilerServices;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
12         ↳ IConverter<TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↳ EqualityComparer<TLink>.Default;
16         private static readonly TLink _zero = default;
17         private static readonly TLink _one = Arithmetic.Increment(_zero);
18
19         private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
23             ↳ powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
24             ↳ powerOf2ToUnaryNumberConverter;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Convert(TLink number)
28         {
29             var links = _links;
30             var nullConstant = links.Constants.Null;
31             var target = nullConstant;
32             for (var i = 0; !_equalityComparer.Equals(number, _zero) && i <
33                 ↳ NumericType<TLink>.BitsSize; i++)
34             {
35                 if (_equalityComparer.Equals(Bit.And(number, _one), _one))
36                 {
37                     target = _equalityComparer.Equals(target, nullConstant)
38                         ? _powerOf2ToUnaryNumberConverter.Convert(i)
39                         : links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
40                 }
41                 number = Bit.ShiftRight(number, 1);
42             }
43             return target;
44         }
45     }
46 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4  using Platform.Converters;
5  using System.Runtime.CompilerServices;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
12         ↳ IConverter<Doublet<TLink>, TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↳ EqualityComparer<TLink>.Default;
16
17         private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
18         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public LinkToItsFrequencyNumberConveter(
22             ILinks<TLink> links,
23             IProperty<TLink, TLink> frequencyPropertyOperator,
24             IConverter<TLink> unaryNumberToAddressConverter)
25             : base(links)
26         {
27             _frequencyPropertyOperator = frequencyPropertyOperator;
28             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public TLink Convert(Doublet<TLink> doublet)

```

```

31 {
32     var links = _links;
33     var link = links.SearchOrDefault(douplet.Source, doublet.Target);
34     if (_equalityComparer.Equals(link, default))
35     {
36         throw new ArgumentException($"Link ({douplet}) not found.", nameof(douplet));
37     }
38     var frequency = _frequencyPropertyOperator.Get(link);
39     if (_equalityComparer.Equals(frequency, default))
40     {
41         return default;
42     }
43     var frequencyNumber = links.GetSource(frequency);
44     return _unaryNumberToAddressConverter.Convert(frequencyNumber);
45 }
46 }
47 }

```

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

```

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

```

1.62 ./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     }
20 }

```

```

15     private static readonly TLink _zero = default;
16     private static readonly TLink _one = Arithmetic.Increment(_zero);
17
18     private readonly Dictionary<TLink, TLink> _unaryToUInt64;
19     private readonly TLink _unaryOne;
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
23         : base(links)
24     {
25         _unaryOne = unaryOne;
26         _unaryToUInt64 = CreateUnaryToUInt64Dictionary(links, unaryOne);
27     }
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public TLink Convert(TLink unaryNumber)
31     {
32         if (_equalityComparer.Equals(unaryNumber, default))
33         {
34             return default;
35         }
36         if (_equalityComparer.Equals(unaryNumber, _unaryOne))
37         {
38             return _one;
39         }
40         var links = _links;
41         var source = links.GetSource(unaryNumber);
42         var target = links.GetTarget(unaryNumber);
43         if (_equalityComparer.Equals(source, target))
44         {
45             return _unaryToUInt64[unaryNumber];
46         }
47         else
48         {
49             var result = _unaryToUInt64[source];
50             TLink lastValue;
51             while (!_unaryToUInt64.TryGetValue(target, out lastValue))
52             {
53                 source = links.GetSource(target);
54                 result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
55                 target = links.GetTarget(target);
56             }
57             result = Arithmetic<TLink>.Add(result, lastValue);
58             return result;
59         }
60     }
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     private static Dictionary<TLink, TLink> CreateUnaryToUInt64Dictionary(ILinks<TLink>
64     ↪ links, TLink unaryOne)
65     {
66         var unaryToUInt64 = new Dictionary<TLink, TLink>
67         {
68             { unaryOne, _one }
69         };
70         var unary = unaryOne;
71         var number = _one;
72         for (var i = 1; i < 64; i++)
73         {
74             unary = links.GetOrCreate(unary, unary);
75             number = Double(number);
76             unaryToUInt64.Add(unary, number);
77         }
78         return unaryToUInt64;
79     }
80
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     private static TLink Double(TLink number) =>
83     ↪ _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
84 }

```

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

```

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

```

```

8
9 namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
12         ↳ IConverter<TLink>
13     {
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 IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
23             ↳ TLink> powerOf2ToUnaryNumberConverter) : base(links) => _unaryNumberPowerOf2Indicies
24             ↳ = CreateUnaryNumberPowerOf2IndiciesDictionary(powerOf2ToUnaryNumberConverter);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Convert(TLink sourceNumber)
28         {
29             var links = _links;
30             var nullConstant = links.Constants.Null;
31             var source = sourceNumber;
32             var target = nullConstant;
33             if (!_equalityComparer.Equals(source, nullConstant))
34             {
35                 while (true)
36                 {
37                     if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
38                     {
39                         SetBit(ref target, powerOf2Index);
40                         break;
41                     }
42                     else
43                     {
44                         powerOf2Index = _unaryNumberPowerOf2Indicies[links.GetSource(source)];
45                         SetBit(ref target, powerOf2Index);
46                         source = links.GetTarget(source);
47                     }
48                 }
49             }
50             return target;
51         }
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         private static Dictionary<TLink, int>
55             ↳ CreateUnaryNumberPowerOf2IndiciesDictionary(IConverter<int, TLink>
56             ↳ powerOf2ToUnaryNumberConverter)
57         {
58             var unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
59             for (int i = 0; i < NumericType<TLink>.BitsSize; i++)
60             {
61                 unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
62             }
63             return unaryNumberPowerOf2Indicies;
64         }
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         private static void SetBit(ref TLink target, int powerOf2Index) => target =
68             ↳ Bit.Or(target, Bit.ShiftLeft(_one, powerOf2Index));
69     }
70 }

```

1.64 ./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,
11         ↳ TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↳ EqualityComparer<TLink>.Default;
15     }
16 }

```



```

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 }

```

1.65 ./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 continueConstant = 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 =>

```

```

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

```

1.66 ./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)]

```

```

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

```

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

```

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

```

```

57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
59     ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
    ↪ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
    : base(links)
60 {
61     _baseConverter = baseConverter;
62     _doubletFrequenciesCache = doubletFrequenciesCache;
63     if (_comparer.Compare(minFrequencyToCompress, _one) < 0)
64     {
65         minFrequencyToCompress = _one;
66     }
67     _minFrequencyToCompress = minFrequencyToCompress;
68     _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
69     ResetMaxDoublet();
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 public override TLink Convert(IList<TLink> source) =>
74     ↪ _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.Source = sequence[i - 1];
101         doublet.Target = sequence[i];
102         LinkFrequency<TLink> data;
103         if (_doInitialFrequenciesIncrement)
104         {
105             data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
106         }
107         else
108         {
109             data = _doubletFrequenciesCache.GetFrequency(ref doublet);
110             if (data == null)
111             {
112                 throw new NotSupportedException("If you ask not to increment
113                     ↪ frequencies, it is expected that all frequencies for the sequence
114                     ↪ are prepared.");
115             }
116         }
117         copy[i - 1].Element = sequence[i - 1];
118         copy[i - 1].DoubletData = data;
119         UpdateMaxDoublet(ref doublet, data);
120     }
121     copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
122     copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
123     if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
124     {
125         var newLength = ReplaceDoublets(copy);
126         sequence = new TLink[newLength];
127         for (int i = 0; i < newLength; i++)
128         {
129             sequence[i] = copy[i].Element;
130         }
131     }
132 }

```

```

130     return sequence;
131 }
132
133 /// <remarks>
134 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
135 /// </remarks>
136 [MethodImpl(MethodImplOptions.AggressiveInlining)]
137 private int ReplaceDoublets(HalfDoublet[] copy)
138 {
139     var oldLength = copy.Length;
140     var newLength = copy.Length;
141     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
142     {
143         var maxDoubletSource = _maxDoublet.Source;
144         var maxDoubletTarget = _maxDoublet.Target;
145         if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
146         {
147             _maxDoubletData.Link = _links.GetOrCreate(maxDoubletSource,
148                 ↪ maxDoubletTarget);
149         }
150         var maxDoubletReplacementLink = _maxDoubletData.Link;
151         oldLength--;
152         var oldLengthMinusTwo = oldLength - 1;
153         // Substitute all usages
154         int w = 0, r = 0; // (r == read, w == write)
155         for (; r < oldLength; r++)
156         {
157             if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
158                 ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
159             {
160                 if (r > 0)
161                 {
162                     var previous = copy[w - 1].Element;
163                     copy[w - 1].DoubletData.DecrementFrequency();
164                     copy[w - 1].DoubletData =
165                         ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
166                             ↪ maxDoubletReplacementLink);
167                 }
168                 if (r < oldLengthMinusTwo)
169                 {
170                     var next = copy[r + 2].Element;
171                     copy[r + 1].DoubletData.DecrementFrequency();
172                     copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
173                         ↪ next);
174                 }
175                 copy[w++] = maxDoubletReplacementLink;
176                 r++;
177                 newLength--;
178             }
179             else
180             {
181                 copy[w++] = copy[r];
182             }
183         }
184         if (w < newLength)
185         {
186             copy[w] = copy[r];
187         }
188         oldLength = newLength;
189         ResetMaxDoublet();
190         UpdateMaxDoublet(copy, newLength);
191     }
192     return newLength;
193 }
194
195 [MethodImpl(MethodImplOptions.AggressiveInlining)]
196 private void ResetMaxDoublet()
197 {
198     _maxDoublet = new Doublet<TLink>();
199     _maxDoubletData = new LinkFrequency<TLink>();
200 }
201
202 [MethodImpl(MethodImplOptions.AggressiveInlining)]
203 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
204 {
205     Doublet<TLink> doublet = default;
206     for (var i = 1; i < length; i++)
207     {

```

```

203         doublet.Source = copy[i - 1].Element;
204         doublet.Target = copy[i].Element;
205         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
206     }
207 }
208
209 [MethodImpl(MethodImplOptions.AggressiveInlining)]
210 private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
211 {
212     var frequency = data.Frequency;
213     var maxFrequency = _maxDoubletData.Frequency;
214     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
215     ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
216     ↪ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
217     ↪ _maxDoublet.Target)))
218     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
219     ↪ (_comparer.Compare(maxFrequency, frequency) < 0 ||
220     ↪ (_equalityComparer.Equals(maxFrequency, frequency) &&
221     ↪ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
222     ↪ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
223     ↪ better stability and better compression on sequent data and even on random
224     ↪ numbers data (but gives collisions anyway) */
225     {
226         _maxDoublet = doublet;
227         _maxDoubletData = data;
228     }
229 }
230 }

```

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

```

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

```

1.69 ./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)

```

```

25         : this(links, new SequenceToItsLocalElementLevelsConverter<TLink>(links, new Frequen
    ↪ ciesCacheBasedLinkToItsFrequencyNumberConverter<TLink>(linkFrequenciesCache))) {
    ↪ }
26
27 [MethodImpl(MethodImplOptions.AggressiveInlining)]
28 public OptimalVariantConverter(ILinks<TLink> links)
29     : this(links, new LinkFrequenciesCache<TLink>(links, new
    ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links))) { }
30
31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 public override TLink Convert(ICollection<TLink> sequence)
33 {
34     var length = sequence.Count;
35     if (length == 1)
36     {
37         return sequence[0];
38     }
39     if (length == 2)
40     {
41         return _links.GetOrCreate(sequence[0], sequence[1]);
42     }
43     sequence = sequence.ToArray();
44     var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
45     while (length > 2)
46     {
47         var levelRepeat = 1;
48         var currentLevel = levels[0];
49         var previousLevel = levels[0];
50         var skipOnce = false;
51         var w = 0;
52         for (var i = 1; i < length; i++)
53         {
54             if (_equalityComparer.Equals(currentLevel, levels[i]))
55             {
56                 levelRepeat++;
57                 skipOnce = false;
58                 if (levelRepeat == 2)
59                 {
60                     sequence[w] = _links.GetOrCreate(sequence[i - 1], sequence[i]);
61                     var newLevel = i >= length - 1 ?
62                         GetPreviousLowerThanCurrentOrCurrent(previousLevel,
    ↪ currentLevel) :
63                         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 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Converters;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Sequences.Converters
8 {
9     public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<ILink<TLink>>
10     {
11         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
12
13         private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
    ↪ IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
    ↪ => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public ILink<TLink> Convert(ILink<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 }

```

1.71 ./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 }

```

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

```

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

```

1.73 ./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);

```

```

34         var target = links.GetTarget(cursor);
35         if (_equalityComparer.Equals(_heightProvider.Get(source),
36             ↪ _heightProvider.Get(target)))
37         {
38             break;
39         }
40         else
41         {
42             _stack.Push(source);
43             cursor = target;
44         }
45     }
46     var left = cursor;
47     var right = appendant;
48     while (!_equalityComparer.Equals(cursor = _stack.Pop(), links.Constants.Null))
49     {
50         right = links.GetOrCreate(left, right);
51         left = cursor;
52     }
53     return links.GetOrCreate(left, right);
54 }
55 }

```

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

```

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

```

1.75 ./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     }
33 }

```

```

27 private BitString _visited;
28
29 private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
    ↳ IList<TLink>>>
30 {
31     private readonly IListEqualityComparer<TLink> _listComparer;
32
33     public ItemEquilityComparer() => _listComparer =
    ↳ Default<IListEqualityComparer<TLink>>.Instance;
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
    ↳ KeyValuePair<IList<TLink>, IList<TLink>> right) =>
    ↳ _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
    ↳ right.Value);
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
    ↳ (_listComparer.GetHashCode(pair.Key),
    ↳ _listComparer.GetHashCode(pair.Value)).GetHashCode();
40 }
41
42 private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
43 {
44     private readonly IListComparer<TLink> _listComparer;
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
    ↳ KeyValuePair<IList<TLink>, IList<TLink>> right)
51     {
52         var intermediateResult = _listComparer.Compare(left.Key, right.Key);
53         if (intermediateResult == 0)
54         {
55             intermediateResult = _listComparer.Compare(left.Value, right.Value);
56         }
57         return intermediateResult;
58     }
59 }
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 public DuplicateSegmentsProvider(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>,
    ↳ IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
73     var links = _links;
74     var count = links.Count();
75     _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
76     links.Each(link =>
77     {
78         var linkIndex = links.GetIndex(link);
79         var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
80         var constants = links.Constants;
81         if (!_visited.Get(linkBitIndex))
82         {
83             var sequenceElements = new List<TLink>();
84             var filler = new ListFiller<TLink, TLink>(sequenceElements, constants.Break);
85             _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
    ↳ LinkAddress<TLink>(linkIndex));
86             if (sequenceElements.Count > 2)
87             {
88                 WalkAll(sequenceElements);
89             }
90         }
91         return constants.Continue;
92     });
93     var resultList = _groups.ToList();
94     var comparer = Default<ItemComparer>.Instance;
95     resultList.Sort(comparer);

```

```

96  #if DEBUG
97      foreach (var item in resultList)
98      {
99          PrintDuplicates(item);
100      }
101  #endif
102      return resultList;
103  }
104
105  [MethodImpl(MethodImplOptions.AggressiveInlining)]
106  protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
    ↪ length) => new Segment<TLink>(elements, offset, length);
107
108  [MethodImpl(MethodImplOptions.AggressiveInlining)]
109  protected override void OnDuplicateFound(Segment<TLink> segment)
110  {
111      var duplicates = CollectDuplicatesForSegment(segment);
112      if (duplicates.Count > 1)
113      {
114          _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
    ↪ duplicates));
115      }
116  }
117
118  [MethodImpl(MethodImplOptions.AggressiveInlining)]
119  private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
120  {
121      var duplicates = new List<TLink>();
122      var readAsElement = new HashSet<TLink>();
123      var restrictions = segment.ShiftRight();
124      var constants = _links.Constants;
125      restrictions[0] = constants.Any;
126      _sequences.Each(sequence =>
127      {
128          var sequenceIndex = sequence[constants.IndexPart];
129          duplicates.Add(sequenceIndex);
130          readAsElement.Add(sequenceIndex);
131          return constants.Continue;
132      }, restrictions);
133      if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
134      {
135          return new List<TLink>();
136      }
137      foreach (var duplicate in duplicates)
138      {
139          var duplicateBitIndex = _addressToInt64Converter.Convert(duplicate);
140          _visited.Set(duplicateBitIndex);
141      }
142      if (_sequences is Sequences sequencesExperiments)
143      {
144          var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
    ↪ ashSet<ulong>)(object)readAsElement,
    ↪ (IList<ulong>)segment);
145          foreach (var partiallyMatchedSequence in partiallyMatched)
146          {
147              var sequenceIndex =
    ↪ _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
148              duplicates.Add(sequenceIndex);
149          }
150      }
151      duplicates.Sort();
152      return duplicates;
153  }
154
155  [MethodImpl(MethodImplOptions.AggressiveInlining)]
156  private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
157  {
158      if (!(_links is ILinks<ulong> ulongLinks))
159      {
160          return;
161      }
162      var duplicatesKey = duplicatesItem.Key;
163      var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
164      Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)}");
165      var duplicatesList = duplicatesItem.Value;
166      for (int i = 0; i < duplicatesList.Count; i++)
167      {
168          var sequenceIndex = _addressToUInt64Converter.Convert(duplicatesList[i]);

```

```

169         var formattedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
170             ↳ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
171             ↳ UnicodeMap.IsCharLink(link.Index) ?
172             ↳ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
173         Console.WriteLine(formattedSequenceStructure);
174         var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
175             ↳ ulongLinks);
176         Console.WriteLine(sequenceString);
177     }
178 }

```

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

```

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

```

```

62     }
63
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     public void PrintFrequencies(ICollection<TLink> sequence)
66     {
67         for (var i = 1; i < sequence.Count; i++)
68         {
69             PrintFrequency(sequence[i - 1], sequence[i]);
70         }
71     }
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     public void PrintFrequency(TLink source, TLink target)
75     {
76         var number = GetFrequency(source, target).Frequency;
77         Console.WriteLine("{0},{1} - {2}", source, target, number);
78     }
79
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
82     {
83         if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
84         {
85             data.IncrementFrequency();
86         }
87         else
88         {
89             var link = _links.SearchOrDefault(doublet.Source, doublet.Target);
90             data = new LinkFrequency<TLink>(_one, link);
91             if (!_equalityComparer.Equals(link, default))
92             {
93                 data.Frequency = Arithmetic.Add(data.Frequency,
94                     ↪ _frequencyCounter.Count(link));
95             }
96             _doubletsCache.Add(doublet, data);
97         }
98         return data;
99     }
100
101     [MethodImpl(MethodImplOptions.AggressiveInlining)]
102     public void ValidateFrequencies()
103     {
104         foreach (var entry in _doubletsCache)
105         {
106             var value = entry.Value;
107             var linkIndex = value.Link;
108             if (!_equalityComparer.Equals(linkIndex, default))
109             {
110                 var frequency = value.Frequency;
111                 var count = _frequencyCounter.Count(linkIndex);
112                 // TODO: Why `frequency` always greater than `count` by 1?
113                 if (((_comparer.Compare(frequency, count) > 0) &&
114                     ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count), _one) > 0))
115                     || ((_comparer.Compare(count, frequency) > 0) &&
116                     ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency), _one) > 0)))
117                 {
118                     throw new InvalidOperationException("Frequencies validation failed.");
119                 }
120             }
121             //else
122             // {
123             //     if (value.Frequency > 0)
124             //     {
125             //         var frequency = value.Frequency;
126             //         linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
127             //         var count = _countLinkFrequency(linkIndex);
128             //         if ((frequency > count && frequency - count > 1) || (count > frequency
129             //             ↪ && count - frequency > 1))
130             //             throw new InvalidOperationException("Frequencies validation
131             //             ↪ failed.");
132             //     }
133             // }

```

1.77 ./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 }
```

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

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

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

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

```

21         {
22             return default;
23         }
24         return base.Count();
25     }
26 }
27 }

```

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

```

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

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7  {
8     public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
9     {
10         private readonly ILinks<TLink> _links;

```



```

11     private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
12
13     [MethodImpl(MethodImplOptions.AggressiveInlining)]
14     public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
15         ↪ ICriterionMatcher<TLink> markedSequenceMatcher)
16     {
17         _links = links;
18         _markedSequenceMatcher = markedSequenceMatcher;
19     }
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     public TLink Count(TLink argument) => new
23         ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
24         ↪ _markedSequenceMatcher, argument).Count();
25 }

```

1.82 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.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.Frequencies.Counters
8 {
9     public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
10         ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>
11     {
12         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
16             ↪ ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
17             : base(links, symbol)
18             => _markedSequenceMatcher = markedSequenceMatcher;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override void CountSequenceSymbolFrequency(TLink link)
22         {
23             var symbolFrequencyCounter = new
24                 ↪ MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
25                 ↪ _markedSequenceMatcher, link, _symbol);
26             _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
27         }
28     }
29 }

```

1.83 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.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 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 }

```

1.84 ./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs

```

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     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     protected virtual void CountSequenceSymbolFrequency(TLink link)
57     {
58         var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
59             ↪ link, _symbol);
60         _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
61     }
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     private TLink EachElementHandler(IList<TLink> doublet)
65     {
66         {
67             var constants = _links.Constants;
68             var doubletIndex = doublet[constants.IndexPart];
69             if (_visits.Add(doubletIndex))
70             {
71                 CountCore(doubletIndex);
72             }
73             return constants.Continue;
74         }
75     }
76 }
77
78 }
79
80 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Interfaces;
4 using Platform.Converters;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Sequences.HeightProviders
9 {
10     public class CachedSequenceHeightProvider<TLink> : ISequenceHeightProvider<TLink>
11     {

```

```

12     private static readonly EqualityComparer<TLink> _equalityComparer =
13         ↳ EqualityComparer<TLink>.Default;
14
15     private readonly TLink _heightPropertyMarker;
16     private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
17     private readonly IConverter<TLink> _addressToUnaryNumberConverter;
18     private readonly IConverter<TLink> _unaryNumberToAddressConverter;
19     private readonly IProperties<TLink, TLink, TLink> _propertyOperator;
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     public CachedSequenceHeightProvider(
23         ISequenceHeightProvider<TLink> baseHeightProvider,
24         IConverter<TLink> addressToUnaryNumberConverter,
25         IConverter<TLink> unaryNumberToAddressConverter,
26         TLink heightPropertyMarker,
27         IProperties<TLink, TLink, TLink> propertyOperator)
28     {
29         _heightPropertyMarker = heightPropertyMarker;
30         _baseHeightProvider = baseHeightProvider;
31         _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
32         _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
33         _propertyOperator = propertyOperator;
34     }
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     public TLink Get(TLink sequence)
38     {
39         TLink height;
40         var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
41         if (_equalityComparer.Equals(heightValue, default))
42         {
43             height = _baseHeightProvider.Get(sequence);
44             heightValue = _addressToUnaryNumberConverter.Convert(height);
45             _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
46         }
47         else
48         {
49             height = _unaryNumberToAddressConverter.Convert(heightValue);
50         }
51         return height;
52     }
53 }

```

1.86 ./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 }

```

1.87 ./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 }

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

```

1.89 ./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(ILinks<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(IList<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)]
43         private bool IsIndexedWithIncrement(TLink source, TLink target)
44         {
45             var link = _links.SearchOrDefault(source, target);
46             var indexed = !_equalityComparer.Equals(link, default);
47             if (indexed)
48             {
49                 Increment(link);
50             }
51             return indexed;
52         }
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         private void Increment(TLink link)
56         {
57             var previousFrequency = _frequencyPropertyOperator.Get(link);
58             var frequency = _frequencyIncrementer.Increment(previousFrequency);
59             _frequencyPropertyOperator.Set(link, frequency);
60         }
61     }
62 }

```

1.90 ./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 }

```

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

```

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

```

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

```

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

```

```

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

```

1.93 ./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 }

```

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

```

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

```

```

31         return _sync.ExecuteWriteOperation(() =>
32         {
33             if (sequence.IsNullOrEmpty())
34             {
35                 return Array.Empty<ulong>();
36             }
37             Links.EnsureLinkExists(sequence);
38             if (sequence.Length == 1)
39             {
40                 return sequence;
41             }
42             return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
43         });
44     }
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
48     {
49         #if DEBUG
50             if ((stopAt - startAt) < 0)
51             {
52                 throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
53                     ↪ меньше или равен stopAt");
54             }
55             #endif
56             if ((stopAt - startAt) == 0)
57             {
58                 return new[] { sequence[startAt] };
59             }
60             if ((stopAt - startAt) == 1)
61             {
62                 return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt]) };
63             }
64             var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
65             var last = 0;
66             for (var splitter = startAt; splitter < stopAt; splitter++)
67             {
68                 var left = CreateAllVariants2Core(sequence, startAt, splitter);
69                 var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
70                 for (var i = 0; i < left.Length; i++)
71                 {
72                     for (var j = 0; j < right.Length; j++)
73                     {
74                         var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
75                         if (variant == Constants.Null)
76                         {
77                             throw new NotImplementedException("Creation cancellation is not
78                                 ↪ implemented.");
79                         }
80                         variants[last++] = variant;
81                     }
82                 }
83             }
84             return variants;
85         }
86
87         [MethodImpl(MethodImplOptions.AggressiveInlining)]
88         public List<ulong> CreateAllVariants1(params ulong[] sequence)
89         {
90             return _sync.ExecuteWriteOperation(() =>
91             {
92                 if (sequence.IsNullOrEmpty())
93                 {
94                     return new List<ulong>();
95                 }
96                 Links.Unsync.EnsureLinkExists(sequence);
97                 if (sequence.Length == 1)
98                 {
99                     return new List<ulong> { sequence[0] };
100                 }
101                 var results = new
102                     ↪ List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
103                 return CreateAllVariants1Core(sequence, results);
104             });
105         }
106
107         [MethodImpl(MethodImplOptions.AggressiveInlining)]
108         private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)

```



```

106 {
107     if (sequence.Length == 2)
108     {
109         var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
110         if (link == Constants.Null)
111         {
112             throw new NotImplementedException("Creation cancellation is not
113                 ↳ implemented.");
114         }
115         results.Add(link);
116         return results;
117     }
118     var innerSequenceLength = sequence.Length - 1;
119     var innerSequence = new ulong[innerSequenceLength];
120     for (var li = 0; li < innerSequenceLength; li++)
121     {
122         var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
123         if (link == Constants.Null)
124         {
125             throw new NotImplementedException("Creation cancellation is not
126                 ↳ implemented.");
127         }
128         for (var isi = 0; isi < li; isi++)
129         {
130             innerSequence[isi] = sequence[isi];
131         }
132         innerSequence[li] = link;
133         for (var isi = li + 1; isi < innerSequenceLength; isi++)
134         {
135             innerSequence[isi] = sequence[isi + 1];
136         }
137         CreateAllVariants1Core(innerSequence, results);
138     }
139     return results;
140 }
141
142 #endregion
143
144 [MethodImpl(MethodImplOptions.AggressiveInlining)]
145 public HashSet<ulong> Each1(params ulong[] sequence)
146 {
147     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
148     Each1(link =>
149     {
150         if (!visitedLinks.Contains(link))
151         {
152             visitedLinks.Add(link); // изучить почему случаются повторы
153         }
154         return true;
155     }, sequence);
156     return visitedLinks;
157 }
158
159 [MethodImpl(MethodImplOptions.AggressiveInlining)]
160 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
161 {
162     if (sequence.Length == 2)
163     {
164         Links.Unsync.Each(sequence[0], sequence[1], handler);
165     }
166     else
167     {
168         var innerSequenceLength = sequence.Length - 1;
169         for (var li = 0; li < innerSequenceLength; li++)
170         {
171             var left = sequence[li];
172             var right = sequence[li + 1];
173             if (left == 0 && right == 0)
174             {
175                 continue;
176             }
177             var linkIndex = li;
178             ulong[] innerSequence = null;
179             Links.Unsync.Each(doublet =>
180             {
181                 if (innerSequence == null)
182                 {
183                     innerSequence = new ulong[innerSequenceLength];
184                 }
185             });
186         }
187     }
188 }

```

```

182         for (var isi = 0; isi < linkIndex; isi++)
183         {
184             innerSequence[isi] = sequence[isi];
185         }
186         for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
187         {
188             innerSequence[isi] = sequence[isi + 1];
189         }
190     }
191     innerSequence[linkIndex] = doublet[Constants.IndexPart];
192     Each1(handler, innerSequence);
193     return Constants.Continue;
194 }, Constants.Any, left, right);
195 }
196 }
197 }
198
199 [MethodImpl(MethodImplOptions.AggressiveInlining)]
200 public HashSet<ulong> EachPart(params ulong[] sequence)
201 {
202     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
203     EachPartCore(link =>
204     {
205         var linkIndex = link[Constants.IndexPart];
206         if (!visitedLinks.Contains(linkIndex))
207         {
208             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
209         }
210         return Constants.Continue;
211     }, sequence);
212     return visitedLinks;
213 }
214
215 [MethodImpl(MethodImplOptions.AggressiveInlining)]
216 public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
217 {
218     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
219     EachPartCore(link =>
220     {
221         var linkIndex = link[Constants.IndexPart];
222         if (!visitedLinks.Contains(linkIndex))
223         {
224             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
225             return handler(new LinkAddress<LinkIndex>(linkIndex));
226         }
227         return Constants.Continue;
228     }, sequence);
229 }
230
231 [MethodImpl(MethodImplOptions.AggressiveInlining)]
232 private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
233 ↪ sequence)
234 {
235     if (sequence.IsNullOrEmpty())
236     {
237         return;
238     }
239     Links.EnsureLinkIsAnyOrExists(sequence);
240     if (sequence.Length == 1)
241     {
242         var link = sequence[0];
243         if (link > 0)
244         {
245             handler(new LinkAddress<LinkIndex>(link));
246         }
247         else
248         {
249             Links.Each(Constants.Any, Constants.Any, handler);
250         }
251     }
252     else if (sequence.Length == 2)
253     {
254         //_links.Each(sequence[0], sequence[1], handler);
255         // o_|      x_o ...
256         // x_|      |___|
257         Links.Each(sequence[1], Constants.Any, doublet =>
258         {
259             var match = Links.SearchOrDefault(sequence[0], doublet);

```

```

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.Unsync.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
↪ stepFrom)
312 {
313     var upStep = stepFrom;
314     var firstSource = Links.Unsync.GetTarget(upStep);
315     while (firstSource != right && firstSource != upStep)
316     {
317         upStep = firstSource;
318         firstSource = Links.Unsync.GetSource(upStep);
319     }
320     if (firstSource == right)
321     {
322         handler(new LinkAddress<LinkIndex>(stepFrom));
323     }
324 }
325
326 // TODO: Test
327 [MethodImpl(MethodImplOptions.AggressiveInlining)]
328 private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
329 {
330     Links.Unsync.Each(right, Constants.Any, doublet =>
331     {
332         StepLeft(handler, left, doublet);
333         if (right != doublet)
334         {
335             PartialStepLeft(handler, left, doublet);

```

```

336     }
337     return true;
338 });
339 }
340
341 [MethodImpl(MethodImplOptions.AggressiveInlining)]
342 private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
343 {
344     Links.Unsync.Each(Constants.Any, right, leftStep =>
345     {
346         TryStepLeftUp(handler, left, leftStep);
347         return true;
348     });
349 }
350
351 [MethodImpl(MethodImplOptions.AggressiveInlining)]
352 private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
353 {
354     var upStep = stepFrom;
355     var firstTarget = Links.Unsync.GetSource(upStep);
356     while (firstTarget != left && firstTarget != upStep)
357     {
358         upStep = firstTarget;
359         firstTarget = Links.Unsync.GetTarget(upStep);
360     }
361     if (firstTarget == left)
362     {
363         handler(new LinkAddress<LinkIndex>(stepFrom));
364     }
365 }
366
367 [MethodImpl(MethodImplOptions.AggressiveInlining)]
368 private bool StartsWith(ulong sequence, ulong link)
369 {
370     var upStep = sequence;
371     var firstSource = Links.Unsync.GetSource(upStep);
372     while (firstSource != link && firstSource != upStep)
373     {
374         upStep = firstSource;
375         firstSource = Links.Unsync.GetSource(upStep);
376     }
377     return firstSource == link;
378 }
379
380 [MethodImpl(MethodImplOptions.AggressiveInlining)]
381 private bool EndsWith(ulong sequence, ulong link)
382 {
383     var upStep = sequence;
384     var lastTarget = Links.Unsync.GetTarget(upStep);
385     while (lastTarget != link && lastTarget != upStep)
386     {
387         upStep = lastTarget;
388         lastTarget = Links.Unsync.GetTarget(upStep);
389     }
390     return lastTarget == link;
391 }
392
393 [MethodImpl(MethodImplOptions.AggressiveInlining)]
394 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
395 {
396     return _sync.ExecuteReadOperation(() =>
397     {
398         var results = new List<ulong>();
399         if (sequence.Length > 0)
400         {
401             Links.EnsureLinkExists(sequence);
402             var firstElement = sequence[0];
403             if (sequence.Length == 1)
404             {
405                 results.Add(firstElement);
406                 return results;
407             }
408             if (sequence.Length == 2)
409             {
410                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
411                 if (doublet != Constants.Null)
412                 {
413                     results.Add(doublet);
414                 }

```

```

415         return results;
416     }
417     var linksInSequence = new HashSet<ulong>(sequence);
418     void handler(ICollection<LinkIndex> result)
419     {
420         var resultIndex = result[Links.Constants.IndexPart];
421         var filterPosition = 0;
422         StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
423             ↪ Links.Unsync.GetTarget,
424             x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
425             ↪ x =>
426             {
427                 if (filterPosition == sequence.Length)
428                 {
429                     filterPosition = -2; // Длиннее чем нужно
430                     return false;
431                 }
432                 if (x != sequence[filterPosition])
433                 {
434                     filterPosition = -1;
435                     return false; // Начинается иначе
436                 }
437                 filterPosition++;
438                 return true;
439             });
440         if (filterPosition == sequence.Length)
441         {
442             results.Add(resultIndex);
443         }
444     }
445     if (sequence.Length >= 2)
446     {
447         StepRight(handler, sequence[0], sequence[1]);
448     }
449     var last = sequence.Length - 2;
450     for (var i = 1; i < last; i++)
451     {
452         PartialStepRight(handler, sequence[i], sequence[i + 1]);
453     }
454     if (sequence.Length >= 3)
455     {
456         StepLeft(handler, sequence[sequence.Length - 2],
457             ↪ sequence[sequence.Length - 1]);
458     }
459     }
460     return results;
461 });
462
463 [MethodImpl(MethodImplOptions.AggressiveInlining)]
464 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
465 {
466     return _sync.ExecuteReadOperation(() =>
467     {
468         var results = new HashSet<ulong>();
469         if (sequence.Length > 0)
470         {
471             Links.EnsureLinkExists(sequence);
472             var firstElement = sequence[0];
473             if (sequence.Length == 1)
474             {
475                 results.Add(firstElement);
476                 return results;
477             }
478             if (sequence.Length == 2)
479             {
480                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
481                 if (doublet != Constants.Null)
482                 {
483                     results.Add(doublet);
484                 }
485                 return results;
486             }
487             var matcher = new Matcher(this, sequence, results, null);
488             if (sequence.Length >= 2)
489             {
490                 StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);

```

```

490     }
491     var last = sequence.Length - 2;
492     for (var i = 1; i < last; i++)
493     {
494         PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
495             ↪ sequence[i + 1]);
496     }
497     if (sequence.Length >= 3)
498     {
499         StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
500             ↪ sequence[sequence.Length - 1]);
501     }
502     return results;
503 });
504 }
505
506 public const int MaxSequenceFormatSize = 200;
507
508 [MethodImpl(MethodImplOptions.AggressiveInlining)]
509 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
510     ↪ => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
511
512 [MethodImpl(MethodImplOptions.AggressiveInlining)]
513 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
514     ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
515     ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
516     ↪ elementToString, insertComma, knownElements));
517
518 [MethodImpl(MethodImplOptions.AggressiveInlining)]
519 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
520     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
521     ↪ LinkIndex[] knownElements)
522 {
523     var linksInSequence = new HashSet<ulong>(knownElements);
524     //var entered = new HashSet<ulong>();
525     var sb = new StringBuilder();
526     sb.Append('{');
527     if (links.Exists(sequenceLink))
528     {
529         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
530             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
531             ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
532         {
533             if (insertComma && sb.Length > 1)
534             {
535                 sb.Append(',');
536             }
537             //if (entered.Contains(element))
538             //if {
539             //    sb.Append('{');
540             //    elementToString(sb, element);
541             //    sb.Append('}');
542             //}
543             //else
544             elementToString(sb, element);
545             if (sb.Length < MaxSequenceFormatSize)
546             {
547                 return true;
548             }
549             sb.Append(insertComma ? ", ..." : "...");
550             return false;
551         });
552     }
553     sb.Append('}');
554     return sb.ToString();
555 }
556
557 [MethodImpl(MethodImplOptions.AggressiveInlining)]
558 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
559     ↪ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
560     ↪ knownElements);
561
562 [MethodImpl(MethodImplOptions.AggressiveInlining)]
563 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
564     ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
565     ↪ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
566     ↪ sequenceLink, elementToString, insertComma, knownElements));

```

```

554 [MethodImpl(MethodImplOptions.AggressiveInlining)]
555 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
556     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
557     ↪ LinkIndex[] knownElements)
558 {
559     var linksInSequence = new HashSet<ulong>(knownElements);
560     var entered = new HashSet<ulong>();
561     var sb = new StringBuilder();
562     sb.Append('{');
563     if (links.Exists(sequenceLink))
564     {
565         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
566             x => linksInSequence.Contains(x) || links.IsFullPoint(x),
567             ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
568             {
569                 if (insertComma && sb.Length > 1)
570                 {
571                     sb.Append(',');
572                 }
573                 if (entered.Contains(element))
574                 {
575                     sb.Append('{');
576                     elementToString(sb, element);
577                     sb.Append('}');
578                 }
579                 else
580                 {
581                     elementToString(sb, element);
582                 }
583                 if (sb.Length < MaxSequenceFormatSize)
584                 {
585                     return true;
586                 }
587                 sb.Append(insertComma ? ", ..." : "...");
588                 return false;
589             });
590     }
591     sb.Append('}');
592     return sb.ToString();
593 }
594 [MethodImpl(MethodImplOptions.AggressiveInlining)]
595 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
596 {
597     return _sync.ExecuteReadOperation(() =>
598     {
599         if (sequence.Length > 0)
600         {
601             Links.EnsureLinkExists(sequence);
602             var results = new HashSet<ulong>();
603             for (var i = 0; i < sequence.Length; i++)
604             {
605                 AllUsagesCore(sequence[i], results);
606             }
607             var filteredResults = new List<ulong>();
608             var linksInSequence = new HashSet<ulong>(sequence);
609             foreach (var result in results)
610             {
611                 var filterPosition = -1;
612                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
613                     ↪ Links.Unsync.GetTarget,
614                     x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
615                     ↪ x =>
616                     {
617                         if (filterPosition == (sequence.Length - 1))
618                         {
619                             return false;
620                         }
621                         if (filterPosition >= 0)
622                         {
623                             if (x == sequence[filterPosition + 1])
624                             {
625                                 filterPosition++;
626                             }
627                             else
628                             {
629                                 return false;
630                             }
631                         }
632                     }
633             }
634         }
635     });
636 }

```

```

627         }
628     }
629     if (filterPosition < 0)
630     {
631         if (x == sequence[0])
632         {
633             filterPosition = 0;
634         }
635     }
636     return true;
637 });
638 if (filterPosition == (sequence.Length - 1))
639 {
640     filteredResults.Add(result);
641 }
642 }
643 return filteredResults;
644 }
645 return new List<ulong>();
646 });
647 }
648
649 [MethodImpl(MethodImplOptions.AggressiveInlining)]
650 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
651 {
652     return _sync.ExecuteReadOperation(() =>
653     {
654         if (sequence.Length > 0)
655         {
656             Links.EnsureLinkExists(sequence);
657             var results = new HashSet<ulong>();
658             for (var i = 0; i < sequence.Length; i++)
659             {
660                 AllUsagesCore(sequence[i], results);
661             }
662             var filteredResults = new HashSet<ulong>();
663             var matcher = new Matcher(this, sequence, filteredResults, null);
664             matcher.AddAllPartialMatchedToResults(results);
665             return filteredResults;
666         }
667         return new HashSet<ulong>();
668     });
669 }
670
671 [MethodImpl(MethodImplOptions.AggressiveInlining)]
672 public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
673 ↪ params ulong[] sequence)
674 {
675     return _sync.ExecuteReadOperation(() =>
676     {
677         if (sequence.Length > 0)
678         {
679             Links.EnsureLinkExists(sequence);
680
681             var results = new HashSet<ulong>();
682             var filteredResults = new HashSet<ulong>();
683             var matcher = new Matcher(this, sequence, filteredResults, handler);
684             for (var i = 0; i < sequence.Length; i++)
685             {
686                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
687                 {
688                     return false;
689                 }
690             }
691             return true;
692         }
693         return true;
694     });
695 }
696
697 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
698 //{
699 //    return Sync.ExecuteReadOperation(() =>
700 //    {
701 //        if (sequence.Length > 0)
702 //        {
703 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
704 //
705 //            var firstResults = new HashSet<ulong>();

```



```

705         //         var lastResults = new HashSet<ulong>();
706
707         //         var first = sequence.First(x => x != LinksConstants.Any);
708         //         var last = sequence.Last(x => x != LinksConstants.Any);
709
710         //         AllUsagesCore(first, firstResults);
711         //         AllUsagesCore(last, lastResults);
712
713         //         firstResults.IntersectWith(lastResults);
714
715         //         //for (var i = 0; i < sequence.Length; i++)
716         //         //     AllUsagesCore(sequence[i], results);
717
718         //         var filteredResults = new HashSet<ulong>();
719         //         var matcher = new Matcher(this, sequence, filteredResults, null);
720         //         matcher.AddAllPartialMatchedToResults(firstResults);
721         //         return filteredResults;
722     }
723
724     //     return new HashSet<ulong>();
725     // });
726 //}
727
728 [MethodImpl(MethodImplOptions.AggressiveInlining)]
729 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
730 {
731     return _sync.ExecuteReadOperation((Func<HashSet<ulong>>)((() =>
732     {
733         if (sequence.Length > 0)
734         {
735             ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links,
736                 ↪ (IList<ulong>)sequence);
737             var firstResults = new HashSet<ulong>();
738             var lastResults = new HashSet<ulong>();
739             var first = sequence.First(x => x != Constants.Any);
740             var last = sequence.Last(x => x != Constants.Any);
741             AllUsagesCore(first, firstResults);
742             AllUsagesCore(last, lastResults);
743             firstResults.IntersectWith(lastResults);
744             //for (var i = 0; i < sequence.Length; i++)
745             //     AllUsagesCore(sequence[i], results);
746             var filteredResults = new HashSet<ulong>();
747             var matcher = new Matcher(this, sequence, filteredResults, null);
748             matcher.AddAllPartialMatchedToResults(firstResults);
749             return filteredResults;
750         }
751         return new HashSet<ulong>();
752     }));
753 }
754
755 [MethodImpl(MethodImplOptions.AggressiveInlining)]
756 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
757     ↪ IList<ulong> sequence)
758 {
759     return _sync.ExecuteReadOperation(() =>
760     {
761         if (sequence.Count > 0)
762         {
763             Links.EnsureLinkExists(sequence);
764             var results = new HashSet<LinkIndex>();
765             //var nextResults = new HashSet<ulong>();
766             //for (var i = 0; i < sequence.Length; i++)
767             //{
768             //     AllUsagesCore(sequence[i], nextResults);
769             //     if (results.IsNullOrEmpty())
770             //     {
771             //         results = nextResults;
772             //         nextResults = new HashSet<ulong>();
773             //     }
774             //     else
775             //     {
776             //         results.IntersectWith(nextResults);
777             //         nextResults.Clear();
778             //     }
779             // }
780             //}
781             var collector1 = new AllUsagesCollector1(Links.Unsync, results);
782             collector1.Collect(Links.Unsync.GetLink(sequence[0]));
783             var next = new HashSet<ulong>();

```

```

781     for (var i = 1; i < sequence.Count; i++)
782     {
783         var collector = new AllUsagesCollector1(Links.Unsync, next);
784         collector.Collect(Links.Unsync.GetLink(sequence[i]));
785
786         results.IntersectWith(next);
787         next.Clear();
788     }
789     var filteredResults = new HashSet<ulong>();
790     var matcher = new Matcher(this, sequence, filteredResults, null,
791         ↪ readAsElements);
792     matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
793         ↪ x)); // OrderBy is a Hack
794     return filteredResults;
795 }
796 return new HashSet<ulong>();
797 });
798 }
799
800 // Does not work
801 //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
802 //    ↪ params ulong[] sequence)
803 //{
804 //    var visited = new HashSet<ulong>();
805 //    var results = new HashSet<ulong>();
806 //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
807 //    ↪ true; }, readAsElements);
808 //    var last = sequence.Length - 1;
809 //    for (var i = 0; i < last; i++)
810 //    {
811 //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
812 //    }
813 //    return results;
814 //}
815
816 [MethodImpl(MethodImplOptions.AggressiveInlining)]
817 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
818 {
819     return _sync.ExecuteReadOperation(() =>
820     {
821         if (sequence.Length > 0)
822         {
823             Links.EnsureLinkExists(sequence);
824             //var firstElement = sequence[0];
825             //if (sequence.Length == 1)
826             //{
827             //    //results.Add(firstElement);
828             //    return results;
829             //}
830             //if (sequence.Length == 2)
831             //{
832             //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
833             //    //if (doublet != Doublets.Links.Null)
834             //    //    results.Add(doublet);
835             //    return results;
836             //}
837             //var lastElement = sequence[sequence.Length - 1];
838             //Func<ulong, bool> handler = x =>
839             //{
840             //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
841             //    ↪ results.Add(x);
842             //    return true;
843             //};
844             //if (sequence.Length >= 2)
845             //    StepRight(handler, sequence[0], sequence[1]);
846             //var last = sequence.Length - 2;
847             //for (var i = 1; i < last; i++)
848             //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
849             //if (sequence.Length >= 3)
850             //    StepLeft(handler, sequence[sequence.Length - 2],
851             //    ↪ sequence[sequence.Length - 1]);
852             //if (sequence.Length == 1)
853             //{
854             //    throw new NotImplementedException(); // all sequences, containing
855             //    ↪ this element?
856             //}
857             //if (sequence.Length == 2)

```

```

851         {
852             var results = new List<ulong>();
853             PartialStepRight(results.Add, sequence[0], sequence[1]);
854             return results;
855         }
856         var matches = new List<List<ulong>>();
857         var last = sequence.Length - 1;
858         for (var i = 0; i < last; i++)
859         {
860             var results = new List<ulong>();
861             //StepRight(results.Add, sequence[i], sequence[i + 1]);
862             PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
863             if (results.Count > 0)
864                 matches.Add(results);
865             else
866                 return results;
867             if (matches.Count == 2)
868             {
869                 var merged = new List<ulong>();
870                 for (var j = 0; j < matches[0].Count; j++)
871                     for (var k = 0; k < matches[1].Count; k++)
872                         CloseInnerConnections(merged.Add, matches[0][j],
873 → matches[1][k]);
874                 if (merged.Count > 0)
875                     matches = new List<List<ulong>> { merged };
876                 else
877                     return new List<ulong>();
878             }
879             if (matches.Count > 0)
880             {
881                 var usages = new HashSet<ulong>();
882                 for (int i = 0; i < sequence.Length; i++)
883                 {
884                     AllUsagesCore(sequence[i], usages);
885                 }
886                 //for (int i = 0; i < matches[0].Count; i++)
887                 //    AllUsagesCore(matches[0][i], usages);
888                 //usages.UnionWith(matches[0]);
889                 return usages.ToList();
890             }
891             var firstLinkUsages = new HashSet<ulong>();
892             AllUsagesCore(sequence[0], firstLinkUsages);
893             firstLinkUsages.Add(sequence[0]);
894             //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
895             → sequence[0] }; // or all sequences, containing this element?
896             //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
897             → 1).ToList();
898             var results = new HashSet<ulong>();
899             foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
900             → firstLinkUsages, 1))
901             {
902                 AllUsagesCore(match, results);
903             }
904             return results.ToList();
905         }
906         return new List<ulong>();
907     });
908 }
909
910 /// <remarks>
911 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
912 /// </remarks>
913 [MethodImpl(MethodImplOptions.AggressiveInlining)]
914 public HashSet<ulong> AllUsages(ulong link)
915 {
916     return _sync.ExecuteReadOperation(() =>
917     {
918         var usages = new HashSet<ulong>();
919         AllUsagesCore(link, usages);
920         return usages;
921     });
922 }
923
924 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
925 → той связи с которой начинался поиск (STTTSSSTT),
926 // причём достаточно одного бита для хранения перехода влево или вправо

```

```

923 [MethodImpl(MethodImplOptions.AggressiveInlining)]
924 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
925 {
926     bool handler(ulong doublet)
927     {
928         if (usages.Add(doublet))
929         {
930             AllUsagesCore(doublet, usages);
931         }
932         return true;
933     }
934     Links.Unsync.Each(link, Constants.Any, handler);
935     Links.Unsync.Each(Constants.Any, link, handler);
936 }
937
938 [MethodImpl(MethodImplOptions.AggressiveInlining)]
939 public HashSet<ulong> AllBottomUsages(ulong link)
940 {
941     return _sync.ExecuteReadOperation(() =>
942     {
943         var visits = new HashSet<ulong>();
944         var usages = new HashSet<ulong>();
945         AllBottomUsagesCore(link, visits, usages);
946         return usages;
947     });
948 }
949
950 [MethodImpl(MethodImplOptions.AggressiveInlining)]
951 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
↪ usages)
952 {
953     bool handler(ulong doublet)
954     {
955         if (visits.Add(doublet))
956         {
957             AllBottomUsagesCore(doublet, visits, usages);
958         }
959         return true;
960     }
961     if (Links.Unsync.Count(Constants.Any, link) == 0)
962     {
963         usages.Add(link);
964     }
965     else
966     {
967         Links.Unsync.Each(link, Constants.Any, handler);
968         Links.Unsync.Each(Constants.Any, link, handler);
969     }
970 }
971
972 [MethodImpl(MethodImplOptions.AggressiveInlining)]
973 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
974 {
975     if (Options.UseSequenceMarker)
976     {
977         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
↪ Options.MarkedSequenceMatcher, symbol);
978         return counter.Count();
979     }
980     else
981     {
982         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
↪ symbol);
983         return counter.Count();
984     }
985 }
986
987 [MethodImpl(MethodImplOptions.AggressiveInlining)]
988 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
↪ LinkIndex> outerHandler)
989 {
990     bool handler(ulong doublet)
991     {
992         if (usages.Add(doublet))
993         {
994             if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
995             {
996                 return false;

```

```

997         }
998         if (!AllUsagesCore1(doublet, usages, outerHandler))
999         {
1000             return false;
1001         }
1002     }
1003     return true;
1004 }
1005 return Links.Unsync.Each(link, Constants.Any, handler)
1006     && Links.Unsync.Each(Constants.Any, link, handler);
1007 }
1008
1009 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1010 public void CalculateAllUsages(ulong[] totals)
1011 {
1012     var calculator = new AllUsagesCalculator(Links, totals);
1013     calculator.Calculate();
1014 }
1015
1016 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1017 public void CalculateAllUsages2(ulong[] totals)
1018 {
1019     var calculator = new AllUsagesCalculator2(Links, totals);
1020     calculator.Calculate();
1021 }
1022
1023 private class AllUsagesCalculator
1024 {
1025     private readonly SynchronizedLinks<ulong> _links;
1026     private readonly ulong[] _totals;
1027
1028     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1029     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
1030     {
1031         _links = links;
1032         _totals = totals;
1033     }
1034
1035     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1036     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1037         ↪ CalculateCore);
1038
1039     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1040     private bool CalculateCore(ulong link)
1041     {
1042         if (_totals[link] == 0)
1043         {
1044             var total = 1UL;
1045             _totals[link] = total;
1046             var visitedChildren = new HashSet<ulong>();
1047             bool linkCalculator(ulong child)
1048             {
1049                 if (link != child && visitedChildren.Add(child))
1050                 {
1051                     total += _totals[child] == 0 ? 1 : _totals[child];
1052                 }
1053                 return true;
1054             }
1055             _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1056             _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1057             _totals[link] = total;
1058         }
1059         return true;
1060     }
1061 }
1062
1063 private class AllUsagesCalculator2
1064 {
1065     private readonly SynchronizedLinks<ulong> _links;
1066     private readonly ulong[] _totals;
1067
1068     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1069     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1070     {
1071         _links = links;
1072         _totals = totals;
1073     }
1074
1075     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

1075 public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1076     ↪ CalculateCore);
1077
1078 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1079 private bool IsElement(ulong link)
1080 {
1081     // _linksInSequence.Contains(link) ||
1082     return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1083         ↪ link;
1084 }
1085
1086 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1087 private bool CalculateCore(ulong link)
1088 {
1089     // TODO: Проработать защиту от заикливания
1090     // Основано на SequenceWalker.WalkLeft
1091     Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1092     Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1093     Func<ulong, bool> isElement = IsElement;
1094     void visitLeaf(ulong parent)
1095     {
1096         if (link != parent)
1097         {
1098             _totals[parent]++;
1099         }
1100     }
1101     void visitNode(ulong parent)
1102     {
1103         if (link != parent)
1104         {
1105             _totals[parent]++;
1106         }
1107     }
1108     var stack = new Stack();
1109     var element = link;
1110     if (isElement(element))
1111     {
1112         visitLeaf(element);
1113     }
1114     else
1115     {
1116         while (true)
1117         {
1118             if (isElement(element))
1119             {
1120                 if (stack.Count == 0)
1121                 {
1122                     break;
1123                 }
1124                 element = stack.Pop();
1125                 var source = getSource(element);
1126                 var target = getTarget(element);
1127                 // Обработка элемента
1128                 if (isElement(target))
1129                 {
1130                     visitLeaf(target);
1131                 }
1132                 if (isElement(source))
1133                 {
1134                     visitLeaf(source);
1135                 }
1136                 element = source;
1137             }
1138             else
1139             {
1140                 stack.Push(element);
1141                 visitNode(element);
1142                 element = getTarget(element);
1143             }
1144         }
1145         _totals[link]++;
1146         return true;
1147     }
1148 }
1149
1150 private class AllUsagesCollector
1151 {
1152     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 заикливания
1237     }
1238
1239     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1240     public bool Collect(ulong link)
1241     {
1242         if (_enter.Add(link))
1243         {
1244             if (_intersectWith.Contains(link))
1245             {
1246                 _usages.Add(link);
1247             }
1248             _links.Unsync.Each(link, _links.Constants.Any, Collect);
1249             _links.Unsync.Each(_links.Constants.Any, link, Collect);
1250         }
1251         return true;
1252     }
1253 }
1254
1255 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1256 private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
↪ right)
1257 {
1258     TryStepLeftUp(handler, left, right);
1259     TryStepRightUp(handler, right, left);
1260 }
1261
1262 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1263 private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
↪ right)
1264 {
1265     // Direct
1266     if (left == right)
1267     {
1268         handler(new LinkAddress<LinkIndex>(left));
1269     }
1270     var doublet = Links.Unsync.SearchOrDefault(left, right);
1271     if (doublet != Constants.Null)
1272     {
1273         handler(new LinkAddress<LinkIndex>(doublet));
1274     }
1275     // Inner
1276     CloseInnerConnections(handler, left, right);
1277     // Outer
1278     StepLeft(handler, left, right);
1279     StepRight(handler, left, right);
1280     PartialStepRight(handler, left, right);
1281     PartialStepLeft(handler, left, right);
1282 }
1283
1284 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1285 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
↪ HashSet<ulong> previousMatchings, long startAt)
1286 {
1287     if (startAt >= sequence.Length) // ?
1288     {
1289         return previousMatchings;
1290     }
1291     var secondLinkUsages = new HashSet<ulong>();
1292     AllUsagesCore(sequence[startAt], secondLinkUsages);
1293     secondLinkUsages.Add(sequence[startAt]);
1294     var matchings = new HashSet<ulong>();
1295     var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
1296     //for (var i = 0; i < previousMatchings.Count; i++)
1297     foreach (var secondLinkUsage in secondLinkUsages)
1298     {
1299         foreach (var previousMatching in previousMatchings)
1300         {
1301             //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
↪ secondLinkUsage);
1302             StepRight(filler.AddFirstAndReturnConstant, previousMatching,
↪ secondLinkUsage);
1303             TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
↪ previousMatching);

```



```

1304         //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
        ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
        ↪ желаемым результатам.
1305     PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
        ↪ secondLinkUsage);
1306     }
1307 }
1308 if (matchings.Count == 0)
1309 {
1310     return matchings;
1311 }
1312 return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1313 }
1314
1315 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1316 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
        ↪ links, params ulong[] sequence)
1317 {
1318     if (sequence == null)
1319     {
1320         return;
1321     }
1322     for (var i = 0; i < sequence.Length; i++)
1323     {
1324         if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
        ↪ !links.Exists(sequence[i]))
1325         {
1326             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
        ↪ $"patternSequence[{i}]");
1327         }
1328     }
1329 }
1330
1331 // Pattern Matching -> Key To Triggers
1332 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1333 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1334 {
1335     return _sync.ExecuteReadOperation(() =>
1336     {
1337         patternSequence = Simplify(patternSequence);
1338         if (patternSequence.Length > 0)
1339         {
1340             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1341             var uniqueSequenceElements = new HashSet<ulong>();
1342             for (var i = 0; i < patternSequence.Length; i++)
1343             {
1344                 if (patternSequence[i] != Constants.Any && patternSequence[i] !=
        ↪ ZeroOrMany)
1345                 {
1346                     uniqueSequenceElements.Add(patternSequence[i]);
1347                 }
1348             }
1349             var results = new HashSet<ulong>();
1350             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1351             {
1352                 AllUsagesCore(uniqueSequenceElement, results);
1353             }
1354             var filteredResults = new HashSet<ulong>();
1355             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1356             matcher.AddAllPatternMatchedToResults(results);
1357             return filteredResults;
1358         }
1359         return new HashSet<ulong>();
1360     });
1361 }
1362
1363 // Найти все возможные связи между указанным списком связей.
1364 // Находит связи между всеми указанными связями в любом порядке.
1365 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
        ↪ несколько раз в последовательности)
1366 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1367 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1368 {
1369     return _sync.ExecuteReadOperation(() =>
1370     {
1371         var results = new HashSet<ulong>();
1372         if (linksToConnect.Length > 0)

```

```

1373     {
1374         Links.EnsureLinkExists(linksToConnect);
1375         AllUsagesCore(linksToConnect[0], results);
1376         for (var i = 1; i < linksToConnect.Length; i++)
1377         {
1378             var next = new HashSet<ulong>();
1379             AllUsagesCore(linksToConnect[i], next);
1380             results.IntersectWith(next);
1381         }
1382     }
1383     return results;
1384 });
1385 }
1386
1387 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1388 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1389 {
1390     return _sync.ExecuteReadOperation(() =>
1391     {
1392         var results = new HashSet<ulong>();
1393         if (linksToConnect.Length > 0)
1394         {
1395             Links.EnsureLinkExists(linksToConnect);
1396             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1397             collector1.Collect(linksToConnect[0]);
1398             var next = new HashSet<ulong>();
1399             for (var i = 1; i < linksToConnect.Length; i++)
1400             {
1401                 var collector = new AllUsagesCollector(Links.Unsync, next);
1402                 collector.Collect(linksToConnect[i]);
1403                 results.IntersectWith(next);
1404                 next.Clear();
1405             }
1406         }
1407         return results;
1408     });
1409 }
1410
1411 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1412 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1413 {
1414     return _sync.ExecuteReadOperation(() =>
1415     {
1416         var results = new HashSet<ulong>();
1417         if (linksToConnect.Length > 0)
1418         {
1419             Links.EnsureLinkExists(linksToConnect);
1420             var collector1 = new AllUsagesCollector(Links, results);
1421             collector1.Collect(linksToConnect[0]);
1422             //AllUsagesCore(linksToConnect[0], results);
1423             for (var i = 1; i < linksToConnect.Length; i++)
1424             {
1425                 var next = new HashSet<ulong>();
1426                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1427                 collector.Collect(linksToConnect[i]);
1428                 //AllUsagesCore(linksToConnect[i], next);
1429                 //results.IntersectWith(next);
1430                 results = next;
1431             }
1432         }
1433         return results;
1434     });
1435 }
1436
1437 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1438 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1439 {
1440     return _sync.ExecuteReadOperation(() =>
1441     {
1442         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1443         ↪ BitArray((int)_links.Total + 1);
1444         if (linksToConnect.Length > 0)
1445         {
1446             Links.EnsureLinkExists(linksToConnect);
1447             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1448             collector1.Collect(linksToConnect[0]);
1449             for (var i = 1; i < linksToConnect.Length; i++)
1450             {

```

```

1450         var next = new BitString((long)Links.Unsync.Count() + 1); //new
1451         ↪ BitArray((int)_links.Total + 1);
1452         var collector = new AllUsagesCollector2(Links.Unsync, next);
1453         collector.Collect(linksToConnect[i]);
1454         results = results.And(next);
1455     }
1456     return results.GetSetUInt64Indices();
1457 });
1458 }
1459
1460 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1461 private static ulong[] Simplify(ulong[] sequence)
1462 {
1463     // Считаем новый размер последовательности
1464     long newLength = 0;
1465     var zeroOrManyStepped = false;
1466     for (var i = 0; i < sequence.Length; i++)
1467     {
1468         if (sequence[i] == ZeroOrMany)
1469         {
1470             if (zeroOrManyStepped)
1471             {
1472                 continue;
1473             }
1474             zeroOrManyStepped = true;
1475         }
1476         else
1477         {
1478             //if (zeroOrManyStepped) Is it efficient?
1479             zeroOrManyStepped = false;
1480         }
1481         newLength++;
1482     }
1483     // Строим новую последовательность
1484     zeroOrManyStepped = false;
1485     var newSequence = new ulong[newLength];
1486     long j = 0;
1487     for (var i = 0; i < sequence.Length; i++)
1488     {
1489         //var current = zeroOrManyStepped;
1490         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1491         //if (current && zeroOrManyStepped)
1492         //    continue;
1493         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1494         //if (zeroOrManyStepped && newZeroOrManyStepped)
1495         //    continue;
1496         //zeroOrManyStepped = newZeroOrManyStepped;
1497         if (sequence[i] == ZeroOrMany)
1498         {
1499             if (zeroOrManyStepped)
1500             {
1501                 continue;
1502             }
1503             zeroOrManyStepped = true;
1504         }
1505         else
1506         {
1507             //if (zeroOrManyStepped) Is it efficient?
1508             zeroOrManyStepped = false;
1509         }
1510         newSequence[j++] = sequence[i];
1511     }
1512     return newSequence;
1513 }
1514
1515 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1516 public static void TestSimplify()
1517 {
1518     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
1519     ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1520     var simplifiedSequence = Simplify(sequence);
1521 }
1522
1523 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1524 public List<ulong> GetSimilarSequences() => new List<ulong>();
1525
1526 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1527 public void Prediction()

```

```

1527 {
1528     //_links
1529     //_sequences
1530 }
1531
1532 #region From Triplets
1533
1534 //public static void DeleteSequence(Link sequence)
1535 //{
1536 //}
1537
1538 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1539 public List<ulong> CollectMatchingSequences(ulong[] links)
1540 {
1541     if (links.Length == 1)
1542     {
1543         throw new InvalidOperationException("Подпоследовательности с одним элементом не
1544             ↳ поддерживаются.");
1545     }
1546     var leftBound = 0;
1547     var rightBound = links.Length - 1;
1548     var left = links[leftBound++];
1549     var right = links[rightBound--];
1550     var results = new List<ulong>();
1551     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1552     return results;
1553 }
1554
1555 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1556 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1557     ↳ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1558 {
1559     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1560     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1561     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1562     {
1563         var nextLeftLink = middleLinks[leftBound];
1564         var elements = GetRightElements(leftLink, nextLeftLink);
1565         if (leftBound <= rightBound)
1566         {
1567             for (var i = elements.Length - 1; i >= 0; i--)
1568             {
1569                 var element = elements[i];
1570                 if (element != 0)
1571                 {
1572                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
1573                         ↳ rightLink, rightBound, ref results);
1574                 }
1575             }
1576         }
1577         else
1578         {
1579             for (var i = elements.Length - 1; i >= 0; i--)
1580             {
1581                 var element = elements[i];
1582                 if (element != 0)
1583                 {
1584                     results.Add(element);
1585                 }
1586             }
1587         }
1588     }
1589     else
1590     {
1591         var nextRightLink = middleLinks[rightBound];
1592         var elements = GetLeftElements(rightLink, nextRightLink);
1593         if (leftBound <= rightBound)
1594         {
1595             for (var i = elements.Length - 1; i >= 0; i--)
1596             {
1597                 var element = elements[i];
1598                 if (element != 0)
1599                 {
1600                     CollectMatchingSequences(leftLink, leftBound, middleLinks,
1601                         ↳ elements[i], rightBound - 1, ref results);
1602                 }
1603             }
1604         }
1605     }
1606 }

```

```

1601         else
1602         {
1603             for (var i = elements.Length - 1; i >= 0; i--)
1604             {
1605                 var element = elements[i];
1606                 if (element != 0)
1607                 {
1608                     results.Add(element);
1609                 }
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
1670 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1671 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1672 {
1673     var result = new ulong[5];
1674     TryStepLeft(startLink, leftLink, result, 0);
1675     Links.Each(startLink, Constants.Any, couple =>
1676     {
1677         if (couple != startLink)
1678         {

```

```

1678         if (TryStepLeft(couple, leftLink, result, 2))
1679         {
1680             return false;
1681         }
1682     }
1683     return true;
1684 });
1685 if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1686 {
1687     result[4] = leftLink;
1688 }
1689 return result;
1690 }
1691
1692 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1693 public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1694 {
1695     var added = 0;
1696     Links.Each(Constants.Any, startLink, couple =>
1697     {
1698         if (couple != startLink)
1699         {
1700             var coupleSource = Links.GetSource(couple);
1701             if (coupleSource == leftLink)
1702             {
1703                 result[offset] = couple;
1704                 if (++added == 2)
1705                 {
1706                     return false;
1707                 }
1708             }
1709             else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1710                 ↪ == Net.And &&
1711             {
1712                 result[offset + 1] = couple;
1713                 if (++added == 2)
1714                 {
1715                     return false;
1716                 }
1717             }
1718             return true;
1719         });
1720     return added > 0;
1721 }
1722
1723 #endregion
1724
1725 #region Walkers
1726
1727 public class PatternMatcher : RightSequenceWalker<ulong>
1728 {
1729     private readonly Sequences _sequences;
1730     private readonly ulong[] _patternSequence;
1731     private readonly HashSet<LinkIndex> _linksInSequence;
1732     private readonly HashSet<LinkIndex> _results;
1733
1734     #region Pattern Match
1735
1736     enum PatternBlockType
1737     {
1738         Undefined,
1739         Gap,
1740         Elements
1741     }
1742
1743     struct PatternBlock
1744     {
1745         public PatternBlockType Type;
1746         public long Start;
1747         public long Stop;
1748     }
1749
1750     private readonly List<PatternBlock> _pattern;
1751     private int _patternPosition;
1752     private long _sequencePosition;
1753
1754     #endregion
1755
1756     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

1757 public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1758 ↪ HashSet<LinkIndex> results)
1759 : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1760 {
1761     _sequences = sequences;
1762     _patternSequence = patternSequence;
1763     _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1764 ↪ _sequences.Constants.Any && x != ZeroOrMany));
1765     _results = results;
1766     _pattern = CreateDetailedPattern();
1767 }
1768 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1769 protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
1770 ↪ base.IsElement(link);
1771 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1772 public bool PatternMatch(LinkIndex sequenceToMatch)
1773 {
1774     _patternPosition = 0;
1775     _sequencePosition = 0;
1776     foreach (var part in Walk(sequenceToMatch))
1777     {
1778         if (!PatternMatchCore(part))
1779         {
1780             break;
1781         }
1782     }
1783     return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1784 ↪ - 1 && _pattern[_patternPosition].Start == 0);
1785 }
1786 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1787 private List<PatternBlock> CreateDetailedPattern()
1788 {
1789     var pattern = new List<PatternBlock>();
1790     var patternBlock = new PatternBlock();
1791     for (var i = 0; i < _patternSequence.Length; i++)
1792     {
1793         if (patternBlock.Type == PatternBlockType.Undefined)
1794         {
1795             if (_patternSequence[i] == _sequences.Constants.Any)
1796             {
1797                 patternBlock.Type = PatternBlockType.Gap;
1798                 patternBlock.Start = 1;
1799                 patternBlock.Stop = 1;
1800             }
1801             else if (_patternSequence[i] == ZeroOrMany)
1802             {
1803                 patternBlock.Type = PatternBlockType.Gap;
1804                 patternBlock.Start = 0;
1805                 patternBlock.Stop = long.MaxValue;
1806             }
1807             else
1808             {
1809                 patternBlock.Type = PatternBlockType.Elements;
1810                 patternBlock.Start = i;
1811                 patternBlock.Stop = i;
1812             }
1813         }
1814         else if (patternBlock.Type == PatternBlockType.Elements)
1815         {
1816             if (_patternSequence[i] == _sequences.Constants.Any)
1817             {
1818                 pattern.Add(patternBlock);
1819                 patternBlock = new PatternBlock
1820                 {
1821                     Type = PatternBlockType.Gap,
1822                     Start = 1,
1823                     Stop = 1
1824                 };
1825             }
1826             else if (_patternSequence[i] == ZeroOrMany)
1827             {
1828                 pattern.Add(patternBlock);
1829                 patternBlock = new PatternBlock
1830                 {
1831                     Type = PatternBlockType.Gap,
1832                     Start = 0,
1833                     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;

```



```

1911 // element = 0;
1912 // for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1913 // {
1914 //     if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1915 //         mininumGap++;
1916 //     else if (_patternSequence[_patternPosition] == ZeroOrMany)
1917 //         maximumGap = long.MaxValue;
1918 //     else
1919 //         break;
1920 // }
1921
1922 // if (maximumGap < mininumGap)
1923 //     maximumGap = mininumGap;
1924 //}
1925
1926 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1927 private bool PatternMatchCore(LinkIndex element)
1928 {
1929     if (_patternPosition >= _pattern.Count)
1930     {
1931         _patternPosition = -2;
1932         return false;
1933     }
1934     var currentPatternBlock = _pattern[_patternPosition];
1935     if (currentPatternBlock.Type == PatternBlockType.Gap)
1936     {
1937         //var currentMatchingBlockLength = (_sequencePosition -
1938         ↪ _lastMatchedBlockPosition);
1939         if (_sequencePosition < currentPatternBlock.Start)
1940         {
1941             _sequencePosition++;
1942             return true; // Двигаемся дальше
1943         }
1944         // Это последний блок
1945         if (_pattern.Count == _patternPosition + 1)
1946         {
1947             _patternPosition++;
1948             _sequencePosition = 0;
1949             return false; // Полное соответствие
1950         }
1951         else
1952         {
1953             if (_sequencePosition > currentPatternBlock.Stop)
1954             {
1955                 return false; // Соответствие невозможно
1956             }
1957             var nextPatternBlock = _pattern[_patternPosition + 1];
1958             if (_patternSequence[nextPatternBlock.Start] == element)
1959             {
1960                 if (nextPatternBlock.Start < nextPatternBlock.Stop)
1961                 {
1962                     _patternPosition++;
1963                     _sequencePosition = 1;
1964                 }
1965                 else
1966                 {
1967                     _patternPosition += 2;
1968                     _sequencePosition = 0;
1969                 }
1970             }
1971         }
1972     }
1973     else // currentPatternBlock.Type == PatternBlockType.Elements
1974     {
1975         var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1976         if (_patternSequence[patternElementPosition] != element)
1977         {
1978             return false; // Соответствие невозможно
1979         }
1980         if (patternElementPosition == currentPatternBlock.Stop)
1981         {
1982             _patternPosition++;
1983             _sequencePosition = 0;
1984         }
1985         else
1986         {
1987             _sequencePosition++;
1988         }
1989     }
1990 }

```

```

1989         return true;
1990         //if (_patternSequence[_patternPosition] != element)
1991         //    return false;
1992         //else
1993         //{
1994             //    _sequencePosition++;
1995             //    _patternPosition++;
1996             //    return true;
1997         //}
1998         //if (_filterPosition == _patternSequence.Length)
1999         //{
2000             //    _filterPosition = -2; // Длиннее чем нужно
2001             //    return false;
2002         //}
2003         //if (element != _patternSequence[_filterPosition])
2004         //{
2005             //    _filterPosition = -1;
2006             //    return false; // Начинается иначе
2007         //}
2008         //if (_filterPosition == (_patternSequence.Length - 1))
2009         //    return false;
2010         //if (_filterPosition >= 0)
2011         //{
2012             //    if (element == _patternSequence[_filterPosition + 1])
2013             //        _filterPosition++;
2014             //    else
2015             //        return false;
2016         //}
2017         //if (_filterPosition < 0)
2018         //{
2019             //    if (element == _patternSequence[0])
2020             //        _filterPosition = 0;
2021         //}
2022     }
2023 }
2024
2025 [MethodImpl(MethodImplOptions.AggressiveInlining)]
2026 public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
2027 {
2028     foreach (var sequenceToMatch in sequencesToMatch)
2029     {
2030         if (PatternMatch(sequenceToMatch))
2031         {
2032             _results.Add(sequenceToMatch);
2033         }
2034     }
2035 }
2036 }
2037 }
2038
2039 #endregion
2040 }
2041 }

```

1.95 ./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
    ↪ вместе, все числа вместе и т.п.
26  /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
    ↪ графа)
27  ///
28  /// x*y - найти все связи между, в последовательностях любой формы, если не стоит
    ↪ ограничитель на то, что является последовательностью, а что нет,
29  /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
    ↪ порядке.
30  ///
31  /// Рост последовательности слева и справа.
32  /// Поиск со звёздочкой.
33  /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
34  /// так же проблема может быть решена при реализации дистанционных триггеров.
35  /// Нужны ли уникальные указатели вообще?
36  /// Что если обращение к информации будет происходить через содержимое всегда?
37  ///
38  /// Писать тесты.
39  ///
40  ///
41  /// Можно убрать зависимость от конкретной реализации Links,
42  /// на зависимость от абстрактного элемента, который может быть представлен несколькими
    ↪ способами.
43  ///
44  /// Можно ли как-то сделать один общий интерфейс
45  ///
46  ///
47  /// Блокчейн и/или гит для распределённой записи транзакций.
48  ///
49  </remarks>
50  public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
    ↪ (после завершения реализации Sequences)
51  {
52      /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
    ↪ связей.</summary>
53      public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
54
55      public SequencesOptions<LinkIndex> Options { get; }
56      public SynchronizedLinks<LinkIndex> Links { get; }
57      private readonly ISynchronization _sync;
58
59      public LinksConstants<LinkIndex> Constants { get; }
60
61      [MethodImpl(MethodImplOptions.AggressiveInlining)]
62      public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
63      {
64          Links = links;
65          _sync = links.SyncRoot;
66          Options = options;
67          Options.ValidateOptions();
68          Options.InitOptions(Links);
69          Constants = links.Constants;
70      }
71
72      [MethodImpl(MethodImplOptions.AggressiveInlining)]
73      public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
    ↪ SequencesOptions<LinkIndex>()) { }
74
75      [MethodImpl(MethodImplOptions.AggressiveInlining)]
76      public bool IsSequence(LinkIndex sequence)
77      {
78          return _sync.ExecuteReadOperation(() =>
79          {
80              if (Options.UseSequenceMarker)
81              {
82                  return Options.MarkedSequenceMatcher.IsMatched(sequence);
83              }
84              return !Links.Unsync.IsPartialPoint(sequence);
85          });
86      }
87
88      [MethodImpl(MethodImplOptions.AggressiveInlining)]
89      private LinkIndex GetSequenceByElements(LinkIndex sequence)
90      {
91          if (Options.UseSequenceMarker)
92          {
93              return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
94          }
95          return sequence;

```

```

96     }
97
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     private LinkIndex GetSequenceElements(LinkIndex sequence)
100    {
101        if (Options.UseSequenceMarker)
102        {
103            var linkContents = new Link<ulong>(Links.GetLink(sequence));
104            if (linkContents.Source == Options.SequenceMarkerLink)
105            {
106                return linkContents.Target;
107            }
108            if (linkContents.Target == Options.SequenceMarkerLink)
109            {
110                return linkContents.Source;
111            }
112        }
113        return sequence;
114    }
115
116    #region Count
117
118    [MethodImpl(MethodImplOptions.AggressiveInlining)]
119    public LinkIndex Count(IList<LinkIndex> restrictions)
120    {
121        if (restrictions.IsNullOrEmpty())
122        {
123            return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
124        }
125        if (restrictions.Count == 1) // Первая связь это адрес
126        {
127            var sequenceIndex = restrictions[0];
128            if (sequenceIndex == Constants.Null)
129            {
130                return 0;
131            }
132            if (sequenceIndex == Constants.Any)
133            {
134                return Count(null);
135            }
136            if (Options.UseSequenceMarker)
137            {
138                return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
139            }
140            return Links.Exists(sequenceIndex) ? 1UL : 0;
141        }
142        throw new NotImplementedException();
143    }
144
145    [MethodImpl(MethodImplOptions.AggressiveInlining)]
146    private LinkIndex CountUsages(params LinkIndex[] restrictions)
147    {
148        if (restrictions.Length == 0)
149        {
150            return 0;
151        }
152        if (restrictions.Length == 1) // Первая связь это адрес
153        {
154            if (restrictions[0] == Constants.Null)
155            {
156                return 0;
157            }
158            var any = Constants.Any;
159            if (Options.UseSequenceMarker)
160            {
161                var elementsLink = GetSequenceElements(restrictions[0]);
162                var sequenceLink = GetSequenceByElements(elementsLink);
163                if (sequenceLink != Constants.Null)
164                {
165                    return Links.Count(any, sequenceLink) + Links.Count(any, elementsLink) -
166                        1;
167                }
168                return Links.Count(any, elementsLink);
169            }
170            return Links.Count(any, restrictions[0]);
171        }
172        throw new NotImplementedException();
173    }

```

```

174 #endregion
175
176 #region Create
177
178 [MethodImpl(MethodImplOptions.AggressiveInlining)]
179 public LinkIndex Create(ICollection<LinkIndex> restrictions)
180 {
181     return _sync.ExecuteWriteOperation(() =>
182     {
183         if (restrictions.IsNullOrEmpty())
184         {
185             return Constants.Null;
186         }
187         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
188         return CreateCore(restrictions);
189     });
190 }
191
192 [MethodImpl(MethodImplOptions.AggressiveInlining)]
193 private LinkIndex CreateCore(ICollection<LinkIndex> restrictions)
194 {
195     LinkIndex[] sequence = restrictions.SkipFirst();
196     if (Options.UseIndex)
197     {
198         Options.Index.Add(sequence);
199     }
200     var sequenceRoot = default(LinkIndex);
201     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
202     {
203         var matches = Each(restrictions);
204         if (matches.Count > 0)
205         {
206             sequenceRoot = matches[0];
207         }
208     }
209     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
210     {
211         return CompactCore(sequence);
212     }
213     if (sequenceRoot == default)
214     {
215         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
216     }
217     if (Options.UseSequenceMarker)
218     {
219         return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
220     }
221     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
222 }
223
224 #endregion
225
226 #region Each
227
228 [MethodImpl(MethodImplOptions.AggressiveInlining)]
229 public List<LinkIndex> Each(ICollection<LinkIndex> sequence)
230 {
231     var results = new List<LinkIndex>();
232     var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
233     Each(filler.AddFirstAndReturnConstant, sequence);
234     return results;
235 }
236
237 [MethodImpl(MethodImplOptions.AggressiveInlining)]
238 public LinkIndex Each(Func<ICollection<LinkIndex>, LinkIndex> handler, ICollection<LinkIndex>
↪ restrictions)
239 {
240     return _sync.ExecuteReadOperation(() =>
241     {
242         if (restrictions.IsNullOrEmpty())
243         {
244             return Constants.Continue;
245         }
246         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
247         if (restrictions.Count == 1)
248         {
249             var link = restrictions[0];
250             var any = Constants.Any;
251             if (link == any)

```

```

252     {
253         if (Options.UseSequenceMarker)
254         {
255             return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
                ↳ Options.SequenceMarkerLink, any));
256         }
257         else
258         {
259             return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
                ↳ any));
260         }
261     }
262     if (Options.UseSequenceMarker)
263     {
264         var sequenceLinkValues = Links.Unsync.GetLink(link);
265         if (sequenceLinkValues[Constants.SourcePart] ==
                ↳ Options.SequenceMarkerLink)
266         {
267             link = sequenceLinkValues[Constants.TargetPart];
268         }
269     }
270     var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
271     sequence[0] = link;
272     return handler(sequence);
273 }
274 else if (restrictions.Count == 2)
275 {
276     throw new NotImplementedException();
277 }
278 else if (restrictions.Count == 3)
279 {
280     return Links.Unsync.Each(handler, restrictions);
281 }
282 else
283 {
284     var sequence = restrictions.SkipFirst();
285     if (Options.UseIndex && !Options.Index.MightContain(sequence))
286     {
287         return Constants.Break;
288     }
289     return EachCore(handler, sequence);
290 }
291 });
292 }
293
294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
    ↳ values)
296 {
297     var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
298     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
299     ↳ Id.
    Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
    ↳ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
    ↳ matcher.HandleFullMatched;
300     //if (sequence.Length >= 2)
301     if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
302     {
303         return Constants.Break;
304     }
305     var last = values.Count - 2;
306     for (var i = 1; i < last; i++)
307     {
308         if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
            ↳ Constants.Continue)
309         {
310             return Constants.Break;
311         }
312     }
313     if (values.Count >= 3)
314     {
315         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
            ↳ != Constants.Continue)
316         {
317             return Constants.Break;
318         }
319     }
320     return Constants.Continue;

```

```

321 }
322
323 [MethodImpl(MethodImplOptions.AggressiveInlining)]
324 private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↪ left, LinkIndex right)
325 {
326     return Links.Unsync.Each(doublet =>
327     {
328         var doubletIndex = doublet[Constants.IndexPart];
329         if (StepRight(handler, doubletIndex, right) != Constants.Continue)
330         {
331             return Constants.Break;
332         }
333         if (left != doubletIndex)
334         {
335             return PartialStepRight(handler, doubletIndex, right);
336         }
337         return Constants.Continue;
338     }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
339 }
340
341 [MethodImpl(MethodImplOptions.AggressiveInlining)]
342 private LinkIndex StepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↪ LinkIndex right) => Links.Unsync.Each(rightStep => TryStepRightUp(handler, right,
    ↪ rightStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, left,
    ↪ Constants.Any));
343
344 [MethodImpl(MethodImplOptions.AggressiveInlining)]
345 private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↪ right, LinkIndex stepFrom)
346 {
347     var upStep = stepFrom;
348     var firstSource = Links.Unsync.GetTarget(upStep);
349     while (firstSource != right && firstSource != upStep)
350     {
351         upStep = firstSource;
352         firstSource = Links.Unsync.GetSource(upStep);
353     }
354     if (firstSource == right)
355     {
356         return handler(new LinkAddress<LinkIndex>(stepFrom));
357     }
358     return Constants.Continue;
359 }
360
361 [MethodImpl(MethodImplOptions.AggressiveInlining)]
362 private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
    ↪ LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
    ↪ leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
    ↪ right));
363
364 [MethodImpl(MethodImplOptions.AggressiveInlining)]
365 private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
    ↪ left, LinkIndex stepFrom)
366 {
367     var upStep = stepFrom;
368     var firstTarget = Links.Unsync.GetSource(upStep);
369     while (firstTarget != left && firstTarget != upStep)
370     {
371         upStep = firstTarget;
372         firstTarget = Links.Unsync.GetTarget(upStep);
373     }
374     if (firstTarget == left)
375     {
376         return handler(new LinkAddress<LinkIndex>(stepFrom));
377     }
378     return Constants.Continue;
379 }
380
381 #endregion
382
383 #region Update
384
385 [MethodImpl(MethodImplOptions.AggressiveInlining)]
386 public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
387 {
388     var sequence = restrictions.SkipFirst();
389     var newSequence = substitution.SkipFirst();

```

```

390     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
391     {
392         return Constants.Null;
393     }
394     if (sequence.IsNullOrEmpty())
395     {
396         return Create(substitution);
397     }
398     if (newSequence.IsNullOrEmpty())
399     {
400         Delete(restrictions);
401         return Constants.Null;
402     }
403     return _sync.ExecuteWriteOperation((Func<ulong>)() =>
404     {
405         ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
406         Links.EnsureLinkExists(newSequence);
407         return UpdateCore(sequence, newSequence);
408     }));
409 }
410
411 [MethodImpl(MethodImplOptions.AggressiveInlining)]
412 private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence)
413 {
414     LinkIndex bestVariant;
415     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
416         ↪ !sequence.EqualTo(newSequence))
417     {
418         bestVariant = CompactCore(newSequence);
419     }
420     else
421     {
422         bestVariant = CreateCore(newSequence);
423     }
424     // TODO: Check all options only ones before loop execution
425     // Возможно нужно две версии Each, возвращающий фактические последовательности и с
426     ↪ маркером,
427     // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
428     ↪ можно получить имея только фактические последовательности.
429     foreach (var variant in Each(sequence))
430     {
431         if (variant != bestVariant)
432         {
433             UpdateOneCore(variant, bestVariant);
434         }
435     }
436     return bestVariant;
437 }
438
439 [MethodImpl(MethodImplOptions.AggressiveInlining)]
440 private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
441 {
442     if (Options.UseGarbageCollection)
443     {
444         var sequenceElements = GetSequenceElements(sequence);
445         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
446         var sequenceLink = GetSequenceByElements(sequenceElements);
447         var newSequenceElements = GetSequenceElements(newSequence);
448         var newSequenceLink = GetSequenceByElements(newSequenceElements);
449         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
450         {
451             if (sequenceLink != Constants.Null)
452             {
453                 Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
454             }
455             Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
456         }
457         ClearGarbage(sequenceElementsContents.Source);
458         ClearGarbage(sequenceElementsContents.Target);
459     }
460     else
461     {
462         if (Options.UseSequenceMarker)
463         {
464             var sequenceElements = GetSequenceElements(sequence);
465             var sequenceLink = GetSequenceByElements(sequenceElements);
466             var newSequenceElements = GetSequenceElements(newSequence);
467             var newSequenceLink = GetSequenceByElements(newSequenceElements);

```



```

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(ICollection<LinkIndex> restrictions)
490 {
491     _sync.ExecuteWriteOperation(() =>
492     {
493         var sequence = restrictions.SkipFirst();
494         // TODO: Check all options only ones before loop execution
495         foreach (var linkToDelete in Each(sequence))
496         {
497             DeleteOneCore(linkToDelete);
498         }
499     });
500 }
501
502 [MethodImpl(MethodImplOptions.AggressiveInlining)]
503 private void DeleteOneCore(LinkIndex link)
504 {
505     if (Options.UseGarbageCollection)
506     {
507         var sequenceElements = GetSequenceElements(link);
508         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
509         var sequenceLink = GetSequenceByElements(sequenceElements);
510         if (Options.UseCascadeDelete || CountUsages(link) == 0)
511         {
512             if (sequenceLink != Constants.Null)
513             {
514                 Links.Unsync.Delete(sequenceLink);
515             }
516             Links.Unsync.Delete(link);
517         }
518         ClearGarbage(sequenceElementsContents.Source);
519         ClearGarbage(sequenceElementsContents.Target);
520     }
521     else
522     {
523         if (Options.UseSequenceMarker)
524         {
525             var sequenceElements = GetSequenceElements(link);
526             var sequenceLink = GetSequenceByElements(sequenceElements);
527             if (Options.UseCascadeDelete || CountUsages(link) == 0)
528             {
529                 if (sequenceLink != Constants.Null)
530                 {
531                     Links.Unsync.Delete(sequenceLink);
532                 }
533                 Links.Unsync.Delete(link);
534             }
535         }
536         else
537         {
538             if (Options.UseCascadeDelete || CountUsages(link) == 0)
539             {
540                 Links.Unsync.Delete(link);
541             }
542         }
543     }
544 }

```

```

543     }
544 }
545
546 #endregion
547
548 #region Compactification
549
550 [MethodImpl(MethodImplOptions.AggressiveInlining)]
551 public void CompactAll()
552 {
553     _sync.ExecuteWriteOperation(() =>
554     {
555         var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
556         for (int i = 0; i < sequences.Count; i++)
557         {
558             var sequence = this.ToList(sequences[i]);
559             Compact(sequence.ShiftRight());
560         }
561     });
562 }
563
564 /// <remarks>
565 /// bestVariant можно выбирать по максимальному числу использований,
566 /// но балансированный позволяет гарантировать уникальность (если есть возможность,
567 /// гарантировать его использование в других местах).
568 ///
569 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
570 /// </remarks>
571 [MethodImpl(MethodImplOptions.AggressiveInlining)]
572 public LinkIndex Compact(ICollection<LinkIndex> sequence)
573 {
574     return _sync.ExecuteWriteOperation(() =>
575     {
576         if (sequence.IsNullOrEmpty())
577         {
578             return Constants.Null;
579         }
580         Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
581         return CompactCore(sequence);
582     });
583 }
584
585 [MethodImpl(MethodImplOptions.AggressiveInlining)]
586 private LinkIndex CompactCore(ICollection<LinkIndex> sequence) => UpdateCore(sequence,
587     ↪ sequence);
588
589 #endregion
590
591 #region Garbage Collection
592
593 /// <remarks>
594 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
595 ↪ определить извне или в унаследованном классе
596 /// </remarks>
597 [MethodImpl(MethodImplOptions.AggressiveInlining)]
598 private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
599     ↪ !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
600
601 [MethodImpl(MethodImplOptions.AggressiveInlining)]
602 private void ClearGarbage(LinkIndex link)
603 {
604     if (IsGarbage(link))
605     {
606         var contents = new Link<ulong>(Links.GetLink(link));
607         Links.Unsync.Delete(link);
608         ClearGarbage(contents.Source);
609         ClearGarbage(contents.Target);
610     }
611 }
612
613 #endregion
614
615 #region Walkers
616
617 [MethodImpl(MethodImplOptions.AggressiveInlining)]
618 public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
619 {
620     return _sync.ExecuteReadOperation(() =>
621     {

```

```

619     var links = Links.Unsync;
620     foreach (var part in Options.Walker.Walk(sequence))
621     {
622         if (!handler(part))
623         {
624             return false;
625         }
626     }
627     return true;
628 });
629 }
630
631 public class Matcher : RightSequenceWalker<LinkIndex>
632 {
633     private readonly Sequences _sequences;
634     private readonly IList<LinkIndex> _patternSequence;
635     private readonly HashSet<LinkIndex> _linksInSequence;
636     private readonly HashSet<LinkIndex> _results;
637     private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
638     private readonly HashSet<LinkIndex> _readAsElements;
639     private int _filterPosition;
640
641     [MethodImpl(MethodImplOptions.AggressiveInlining)]
642     public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
643         ↳ HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
644         ↳ HashSet<LinkIndex> readAsElements = null)
645         : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
646     {
647         _sequences = sequences;
648         _patternSequence = patternSequence;
649         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
650             ↳ _links.Constants.Any && x != ZeroOrMany));
651         _results = results;
652         _stopableHandler = stopableHandler;
653         _readAsElements = readAsElements;
654     }
655
656     [MethodImpl(MethodImplOptions.AggressiveInlining)]
657     protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
658         ↳ (_readAsElements != null && _readAsElements.Contains(link)) ||
659         ↳ _linksInSequence.Contains(link);
660
661     [MethodImpl(MethodImplOptions.AggressiveInlining)]
662     public bool FullMatch(LinkIndex sequenceToMatch)
663     {
664         _filterPosition = 0;
665         foreach (var part in Walk(sequenceToMatch))
666         {
667             if (!FullMatchCore(part))
668             {
669                 break;
670             }
671         }
672         return _filterPosition == _patternSequence.Count;
673     }
674
675     [MethodImpl(MethodImplOptions.AggressiveInlining)]
676     private bool FullMatchCore(LinkIndex element)
677     {
678         if (_filterPosition == _patternSequence.Count)
679         {
680             _filterPosition = -2; // Длиннее чем нужно
681             return false;
682         }
683         if (_patternSequence[_filterPosition] != _links.Constants.Any
684             && element != _patternSequence[_filterPosition])
685         {
686             _filterPosition = -1;
687             return false; // Начинается/Продолжается иначе
688         }
689         _filterPosition++;
690         return true;
691     }
692
693     [MethodImpl(MethodImplOptions.AggressiveInlining)]
694     public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
695     {
696         var sequenceToMatch = restrictions[_links.Constants.IndexPart];
697         if (FullMatch(sequenceToMatch))
698         {

```

```

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     }
736     return _filterPosition == _patternSequence.Count - 1;
737 }
738
739 [MethodImpl(MethodImplOptions.AggressiveInlining)]
740 private bool PartialMatchCore(LinkIndex element)
741 {
742     if (_filterPosition == (_patternSequence.Count - 1))
743     {
744         return false; // Нашлось
745     }
746     if (_filterPosition >= 0)
747     {
748         if (element == _patternSequence[_filterPosition + 1])
749         {
750             _filterPosition++;
751         }
752         else
753         {
754             _filterPosition = -1;
755         }
756     }
757     if (_filterPosition < 0)
758     {
759         if (element == _patternSequence[0])
760         {
761             _filterPosition = 0;
762         }
763     }
764     return true; // Ищем дальше
765 }
766
767 [MethodImpl(MethodImplOptions.AggressiveInlining)]
768 public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
769 {
770     if (PartialMatch(sequenceToMatch))
771     {
772         _results.Add(sequenceToMatch);
773     }
774 }

```

```

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 }

```

1.96 ./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 ICollection<TLink> sequences, ICollection<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 ICollection<TLink> ToList<TLink>(this ICollection<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 }

```

```
31     }
32 }
```

1.97 ./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> // TODO0: To use type parameter <TLink> the
        ↳ ILinks<TLink> must contain GetConstants function.
19     {
20         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
21
22         public TLink SequenceMarkerLink
23         {
24             [MethodImpl(MethodImplOptions.AggressiveInlining)]
25             get;
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             set;
28         }
29
30         public bool UseCascadeUpdate
31         {
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             get;
34             [MethodImpl(MethodImplOptions.AggressiveInlining)]
35             set;
36         }
37
38         public bool UseCascadeDelete
39         {
40             [MethodImpl(MethodImplOptions.AggressiveInlining)]
41             get;
42             [MethodImpl(MethodImplOptions.AggressiveInlining)]
43             set;
44         }
45
46         public bool UseIndex
47         {
48             [MethodImpl(MethodImplOptions.AggressiveInlining)]
49             get;
50             [MethodImpl(MethodImplOptions.AggressiveInlining)]
51             set;
52         } // TODO0: 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 }
```

```

77
78 public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting
79 {
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     get;
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     set;
84 }
85
86 public bool EnforceSingleSequenceVersionOnWriteBasedOnNew
87 {
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     get;
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     set;
92 }
93
94 public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher
95 {
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     get;
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     set;
100 }
101
102 public IConverter<IList<TLink>, TLink> LinksToSequenceConverter
103 {
104     [MethodImpl(MethodImplOptions.AggressiveInlining)]
105     get;
106     [MethodImpl(MethodImplOptions.AggressiveInlining)]
107     set;
108 }
109
110 public ISequenceIndex<TLink> Index
111 {
112     [MethodImpl(MethodImplOptions.AggressiveInlining)]
113     get;
114     [MethodImpl(MethodImplOptions.AggressiveInlining)]
115     set;
116 }
117
118 public ISequenceWalker<TLink> Walker
119 {
120     [MethodImpl(MethodImplOptions.AggressiveInlining)]
121     get;
122     [MethodImpl(MethodImplOptions.AggressiveInlining)]
123     set;
124 }
125
126 public bool ReadFullSequence
127 {
128     [MethodImpl(MethodImplOptions.AggressiveInlining)]
129     get;
130     [MethodImpl(MethodImplOptions.AggressiveInlining)]
131     set;
132 }
133
134 // TODO: Реализовать компактификацию при чтении
135 //public bool EnforceSingleSequenceVersionOnRead { get; set; }
136 //public bool UseRequestMarker { get; set; }
137 //public bool StoreRequestResults { get; set; }
138
139 [MethodImpl(MethodImplOptions.AggressiveInlining)]
140 public void InitOptions(ISynchronizedLinks<TLink> links)
141 {
142     if (UseSequenceMarker)
143     {
144         if (!_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
145         {
146             SequenceMarkerLink = links.CreatePoint();
147         }
148         else
149         {
150             if (!links.Exists(SequenceMarkerLink))
151             {
152                 var link = links.CreatePoint();
153                 if (!_equalityComparer.Equals(link, SequenceMarkerLink))
154                 {
155                     throw new InvalidOperationException("Cannot recreate sequence marker
156                     ↪ link.");

```

```

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 else
189 {
190     if (LinksToSequenceConverter == null)
191     {
192         LinksToSequenceConverter = balancedVariantConverter;
193     }
194 }
195 if (UseIndex && Index == null)
196 {
197     Index = new SequenceIndex<TLink>(links);
198 }
199 if (Walker == null)
200 {
201     Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
202 }
203 }
204
205 [MethodImpl(MethodImplOptions.AggressiveInlining)]
206 public void ValidateOptions()
207 {
208     if (UseGarbageCollection && !UseSequenceMarker)
209     {
210         throw new NotSupportedException("To use garbage collection UseSequenceMarker
211             ↪ option must be on.");
212     }
213 }
214 }
215 }

```

1.98 ./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 }

```


1.99 ./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 }
```

1.100 ./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)
```

```

31     {
32         var length = 1;
33         var array = new TLink[length];
34         array[0] = sequence;
35         if (_isElement(sequence))
36         {
37             return array;
38         }
39         bool hasElements;
40         do
41         {
42             length *= 2;
43 #if USEARRAYPOOL
44             var nextArray = ArrayPool.Allocate<ulong>(length);
45 #else
46             var nextArray = new TLink[length];
47 #endif
48             hasElements = false;
49             for (var i = 0; i < array.Length; i++)
50             {
51                 var candidate = array[i];
52                 if (_equalityComparer.Equals(array[i], default))
53                 {
54                     continue;
55                 }
56                 var doubletOffset = i * 2;
57                 if (_isElement(candidate))
58                 {
59                     nextArray[doubletOffset] = candidate;
60                 }
61                 else
62                 {
63                     var links = _links;
64                     var link = links.GetLink(candidate);
65                     var linkSource = links.GetSource(link);
66                     var linkTarget = links.GetTarget(link);
67                     nextArray[doubletOffset] = linkSource;
68                     nextArray[doubletOffset + 1] = linkTarget;
69                     if (!hasElements)
70                     {
71                         hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
72                     }
73                 }
74             }
75 #if USEARRAYPOOL
76             if (array.Length > 1)
77             {
78                 ArrayPool.Free(array);
79             }
80 #endif
81             array = nextArray;
82         }
83         while (hasElements);
84         var filledElementsCount = CountFilledElements(array);
85         if (filledElementsCount == array.Length)
86         {
87             return array;
88         }
89         else
90         {
91             return CopyFilledElements(array, filledElementsCount);
92         }
93     }
94
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
97     {
98         var finalArray = new TLink[filledElementsCount];
99         for (int i = 0, j = 0; i < array.Length; i++)
100         {
101             if (!_equalityComparer.Equals(array[i], default))
102             {
103                 finalArray[j] = array[i];
104                 j++;
105             }
106         }
107 #if USEARRAYPOOL
108         ArrayPool.Free(array);
109 #endif
110         return finalArray;

```

```

111     }
112
113     [MethodImpl(MethodImplOptions.AggressiveInlining)]
114     private static int CountFilledElements(TLink[] array)
115     {
116         var count = 0;
117         for (var i = 0; i < array.Length; i++)
118         {
119             if (!_equalityComparer.Equals(array[i], default))
120             {
121                 count++;
122             }
123         }
124         return count;
125     }
126 }
127 }

```

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

```

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

```

1.102 ./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)]

```

```

16     protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
    ↪ isElement) : base(links)
17     {
18         _stack = stack;
19         _isElement = isElement;
20     }
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
    ↪ stack, links.IsPartialPoint) { }
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     public IEnumerable<TLink> Walk(TLink sequence)
27     {
28         _stack.Clear();
29         var element = sequence;
30         if (IsElement(element))
31         {
32             yield return element;
33         }
34         else
35         {
36             while (true)
37             {
38                 if (IsElement(element))
39                 {
40                     if (_stack.IsEmpty)
41                     {
42                         break;
43                     }
44                     element = _stack.Pop();
45                     foreach (var output in WalkContents(element))
46                     {
47                         yield return output;
48                     }
49                     element = GetNextElementAfterPop(element);
50                 }
51                 else
52                 {
53                     _stack.Push(element);
54                     element = GetNextElementAfterPush(element);
55                 }
56             }
57         }
58     }
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected abstract TLink GetNextElementAfterPop(TLink element);
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected abstract TLink GetNextElementAfterPush(TLink element);
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected abstract IEnumerable<TLink> WalkContents(TLink element);
71 }
72 }

```

1.103 ./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 =
    ↪ EqualityComparer<TLink>.Default;
12
13         private readonly TLink _stack;
14
15         public bool IsEmpty
16         {
17             [MethodImpl(MethodImplOptions.AggressiveInlining)]
18             get => _equalityComparer.Equals(Peek(), _stack);

```

```

19     }
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     public Stack(ILinks<TLink> links, TLink stack) : base(links) => _stack = stack;
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     private TLink GetStackMarker() => _links.GetSource(_stack);
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     private TLink GetTop() => _links.GetTarget(_stack);
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     public TLink Peek() => _links.GetTarget(GetTop());
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     public TLink Pop()
35     {
36         var element = Peek();
37         if (!_equalityComparer.Equals(element, _stack))
38         {
39             var top = GetTop();
40             var previousTop = _links.GetSource(top);
41             _links.Update(_stack, GetStackMarker(), previousTop);
42             _links.Delete(top);
43         }
44         return element;
45     }
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
49         ↪ _links.GetOrCreate(GetTop(), element));
50 }

```

1.104 ./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 }

```

1.105 ./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)]
65     public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
66         ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
70         ↳ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
71         ↳ Unsync.Update);
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     public void Delete(IList<TLinkAddress> restrictions) =>
75         ↳ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
76
77     //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
78     //    ↳ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
79     //{
80     //    if (restriction != null && substitution != null &&
81     //        ↳ !substitution.EqualTo(restriction))
82     //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
83     //        ↳ substitution, substitutedHandler, Unsync.Trigger);
84     //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
85     //        ↳ substitutedHandler, Unsync.Trigger);
86     //}
87 }

```

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

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Singletons;
6  using Platform.Data.Doublets.Unicode;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets
11 {
12     public static class UInt64LinksExtensions
13     {

```

```

14 public static readonly LinksConstants<ulong> Constants =
    ↳ Default<LinksConstants<ulong>>.Instance;
15
16 [MethodImpl(MethodImplOptions.AggressiveInlining)]
17 public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
18
19 [MethodImpl(MethodImplOptions.AggressiveInlining)]
20 public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
21 {
22     if (sequence == null)
23     {
24         return false;
25     }
26     var constants = links.Constants;
27     for (var i = 0; i < sequence.Length; i++)
28     {
29         if (sequence[i] == constants.Any)
30         {
31             return true;
32         }
33     }
34     return false;
35 }
36
37 [MethodImpl(MethodImplOptions.AggressiveInlining)]
38 public static string FormatStructure(this ILinks<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,
91                     ↪ appendElement, renderIndex);
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,
108                         ↪ appendElement, renderIndex);
109                 }
110                 sb.Append(' ');
111             }
112             else
113             {
114                 if (renderDebug)
115                 {
116                     sb.Append('*');
117                 }
118                 sb.Append(linkIndex);
119             }
120         }
121         else
122         {
123             if (renderDebug)
124             {
125                 sb.Append('~');
126             }
127             sb.Append(linkIndex);
128         }
129     }
130 }
131 }

```

1.107 ./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     ///
42     Или
43     ///
44     public struct TransitionHeader
45     {
46         public ulong TransactionIdCombined;
47         public ulong TimestampCombined;
48         ///
49         public ulong TransactionId
50         {
51             get
52             {
53                 return (ulong) mask & TransactionIdCombined;
54             }
55         }
56         ///
57         public UniqueTimestamp Timestamp
58         {
59             get
60             {
61                 return (UniqueTimestamp)mask & TransactionIdCombined;
62             }
63         }
64         ///
65         public TransactionItemType Type
66         {
67             get
68             {
69                 // Использовать по одному биту из TransactionId и Timestamp,
70                 // для значения в 2 бита, которое представляет тип операции
71                 throw new NotImplementedException();
72             }
73         }
74     }
75     ///
76     private struct Transition
77     {
78         public TransitionHeader Header;
79         public Link Source;
80         public Link Linker;
81         public Link Target;
82     }
83     ///
84     </remarks>
85     public struct Transition : IEquatable<Transition>
86     {
87         public static readonly long Size = Structure<Transition>.Size;
88
89         public readonly ulong TransactionId;
90         public readonly Link<ulong> Before;
91         public readonly Link<ulong> After;
92         public readonly Timestamp Timestamp;
93
94         [MethodImpl(MethodImplOptions.AggressiveInlining)]
95         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
96             ↪ transactionId, Link<ulong> before, Link<ulong> after)
97         {
98             TransactionId = transactionId;
99             Before = before;
100             After = after;
101             Timestamp = uniqueTimestampFactory.Create();
102         }

```

```

102 [MethodImpl(MethodImplOptions.AggressiveInlining)]
103 public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
104     ↪ transactionId, Link<ulong> before) : this(uniqueTimestampFactory, transactionId,
105     ↪ before, default) { }

106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
108     ↪ transactionId) : this(uniqueTimestampFactory, transactionId, default, default) {
109     ↪ }

110 [MethodImpl(MethodImplOptions.AggressiveInlining)]
111 public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
112     ↪ {After}";

113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
114 public override bool Equals(object obj) => obj is Transition transition ?
115     ↪ Equals(transition) : false;

116 [MethodImpl(MethodImplOptions.AggressiveInlining)]
117 public override int GetHashCode() => (TransactionId, Before, After,
118     ↪ Timestamp).GetHashCode();

119 [MethodImpl(MethodImplOptions.AggressiveInlining)]
120 public bool Equals(Transition other) => TransactionId == other.TransactionId &&
121     ↪ Before == other.Before && After == other.After && Timestamp == other.Timestamp;

122 [MethodImpl(MethodImplOptions.AggressiveInlining)]
123 public static bool operator ==(Transition left, Transition right) =>
124     ↪ left.Equals(right);

125 [MethodImpl(MethodImplOptions.AggressiveInlining)]
126 public static bool operator !=(Transition left, Transition right) => !(left ==
127     ↪ right);
128 }

129 /// <remarks>
130 /// Другие варианты реализации транзакций (атомарности):
131 /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
132     ↪ Target)) и индексов.
133 /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
134     ↪ потребуется решить вопрос
135     ↪ со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
136     ↪ пересечениями идентификаторов.
137 ///
138 /// Где хранить промежуточный список транзакций?
139 ///
140 /// В оперативной памяти:
141 /// Минусы:
142 /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
143     ↪ так как нужно отдельно выделять память под список трансформаций.
144 /// 2. Выделенной оперативной памяти может не хватить, в том случае,
145     ↪ если транзакция использует слишком много трансформаций.
146     ↪ -> Можно использовать жёсткий диск для слишком длинных транзакций.
147     ↪ -> Максимальный размер списка трансформаций можно ограничить / задать
148     ↪ константой.
149 /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
150     ↪ создавая задержку.
151 ///
152 /// На жёстком диске:
153 /// Минусы:
154 /// 1. Длительный отклик, на запись каждой трансформации.
155 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
156     ↪ -> Это может решаться упаковкой/исключением дублирующих операций.
157     ↪ -> Также это может решаться тем, что короткие транзакции вообще
158     ↪ не будут записываться в случае отката.
159 /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
160     ↪ операции (трансформации)
161     ↪ будут записаны в лог.
162 ///
163 /// </remarks>
164 public class Transaction : DisposableBase
165 {
166     private readonly Queue<Transition> _transitions;
167     private readonly UInt64LinksTransactionsLayer _layer;
168     public bool IsCommitted { get; private set; }
169     public bool IsReverted { get; private set; }
170 }

```

```

164 [MethodImpl(MethodImplOptions.AggressiveInlining)]
165 public Transaction(UInt64LinksTransactionsLayer layer)
166 {
167     _layer = layer;
168     if (_layer._currentTransactionId != 0)
169     {
170         throw new NotSupportedException("Nested transactions not supported.");
171     }
172     IsCommitted = false;
173     IsReverted = false;
174     _transitions = new Queue<Transition>();
175     SetCurrentTransaction(layer, this);
176 }
177
178 [MethodImpl(MethodImplOptions.AggressiveInlining)]
179 public void Commit()
180 {
181     EnsureTransactionAllowsWriteOperations(this);
182     while (_transitions.Count > 0)
183     {
184         var transition = _transitions.Dequeue();
185         _layer._transitions.Enqueue(transition);
186     }
187     _layer._lastCommittedTransactionId = _layer._currentTransactionId;
188     IsCommitted = true;
189 }
190
191 [MethodImpl(MethodImplOptions.AggressiveInlining)]
192 private void Revert()
193 {
194     EnsureTransactionAllowsWriteOperations(this);
195     var transitionsToRevert = new Transition[_transitions.Count];
196     _transitions.CopyTo(transitionsToRevert, 0);
197     for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
198     {
199         _layer.RevertTransition(transitionsToRevert[i]);
200     }
201     IsReverted = true;
202 }
203
204 [MethodImpl(MethodImplOptions.AggressiveInlining)]
205 public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
    ↪ Transaction transaction)
206 {
207     layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
208     layer._currentTransactionTransitions = transaction._transitions;
209     layer._currentTransaction = transaction;
210 }
211
212 [MethodImpl(MethodImplOptions.AggressiveInlining)]
213 public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
214 {
215     if (transaction.IsReverted)
216     {
217         throw new InvalidOperationException("Transation is reverted.");
218     }
219     if (transaction.IsCommitted)
220     {
221         throw new InvalidOperationException("Transation is committed.");
222     }
223 }
224
225 [MethodImpl(MethodImplOptions.AggressiveInlining)]
226 protected override void Dispose(bool manual, bool wasDisposed)
227 {
228     if (!wasDisposed && _layer != null && !_layer.Disposable.IsDisposed)
229     {
230         if (!IsCommitted && !IsReverted)
231         {
232             Revert();
233         }
234         _layer.ResetCurrentTransation();
235     }
236 }
237
238
239 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
240
241 private readonly string _logAddress;

```

```

242 private readonly FileStream _log;
243 private readonly Queue<Transition> _transitions;
244 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
245 private Task _transitionsPusher;
246 private Transition _lastCommittedTransition;
247 private ulong _currentTransactionId;
248 private Queue<Transition> _currentTransactionTransitions;
249 private Transaction _currentTransaction;
250 private ulong _lastCommittedTransactionId;
251
252 [MethodImpl(MethodImplOptions.AggressiveInlining)]
253 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
254     : base(links)
255 {
256     if (string.IsNullOrEmpty(logAddress))
257     {
258         throw new ArgumentNullException(nameof(logAddress));
259     }
260     // В первой строке файла хранится последняя закоммиченную транзакцию.
261     // При запуске это используется для проверки удачного закрытия файла лога.
262     // In the first line of the file the last committed transaction is stored.
263     // On startup, this is used to check that the log file is successfully closed.
264     var lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
265     var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
266     if (!lastCommittedTransition.Equals(lastWrittenTransition))
267     {
268         Dispose();
269         throw new NotSupportedException("Database is damaged, autorecovery is not
270             ↳ supported yet.");
271     }
272     if (lastCommittedTransition == default)
273     {
274         FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
275     }
276     _lastCommittedTransition = lastCommittedTransition;
277     // TODO: Think about a better way to calculate or store this value
278     var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
279     _lastCommittedTransactionId = allTransitions.Length > 0 ? allTransitions.Max(x =>
280         ↳ x.TransactionId) : 0;
281     _uniqueTimestampFactory = new UniqueTimestampFactory();
282     _logAddress = logAddress;
283     _log = FileHelpers.Append(logAddress);
284     _transitions = new Queue<Transition>();
285     _transitionsPusher = new Task(TransitionsPusher);
286     _transitionsPusher.Start();
287 }
288
289 [MethodImpl(MethodImplOptions.AggressiveInlining)]
290 public IList<ulong> GetLinkValue(ulong link) => _links.GetLink(link);
291
292 [MethodImpl(MethodImplOptions.AggressiveInlining)]
293 public override ulong Create(IList<ulong> restrictions)
294 {
295     var createdLinkIndex = _links.Create();
296     var createdLink = new Link<ulong>(_links.GetLink(createdLinkIndex));
297     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
298         ↳ default, createdLink));
299     return createdLinkIndex;
300 }
301
302 [MethodImpl(MethodImplOptions.AggressiveInlining)]
303 public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
304 {
305     var linkIndex = restrictions[_constants.IndexPart];
306     var beforeLink = new Link<ulong>(_links.GetLink(linkIndex));
307     linkIndex = _links.Update(restrictions, substitution);
308     var afterLink = new Link<ulong>(_links.GetLink(linkIndex));
309     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
310         ↳ beforeLink, afterLink));
311     return linkIndex;
312 }
313
314 [MethodImpl(MethodImplOptions.AggressiveInlining)]
315 public override void Delete(IList<ulong> restrictions)
316 {
317     var link = restrictions[_constants.IndexPart];
318     var deletedLink = new Link<ulong>(_links.GetLink(link));
319     _links.Delete(link);

```

```

316         CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
317             ↪ deletedLink, default));
318     }
319     [MethodImpl(MethodImplOptions.AggressiveInlining)]
320     private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
321         ↪ _transitions;
322     [MethodImpl(MethodImplOptions.AggressiveInlining)]
323     private void CommitTransition(Transition transition)
324     {
325         if (_currentTransaction != null)
326         {
327             Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
328         }
329         var transitions = GetCurrentTransitions();
330         transitions.Enqueue(transition);
331     }
332     [MethodImpl(MethodImplOptions.AggressiveInlining)]
333     private void RevertTransition(Transition transition)
334     {
335         if (transition.After.IsNull()) // Revert Deletion with Creation
336         {
337             _links.Create();
338         }
339         else if (transition.Before.IsNull()) // Revert Creation with Deletion
340         {
341             _links.Delete(transition.After.Index);
342         }
343         else // Revert Update
344         {
345             _links.Update(new[] { transition.After.Index, transition.Before.Source,
346                 ↪ transition.Before.Target });
347         }
348     }
349     [MethodImpl(MethodImplOptions.AggressiveInlining)]
350     private void ResetCurrentTransation()
351     {
352         _currentTransactionId = 0;
353         _currentTransactionTransitions = null;
354         _currentTransaction = null;
355     }
356     [MethodImpl(MethodImplOptions.AggressiveInlining)]
357     private void PushTransitions()
358     {
359         if (_log == null || _transitions == null)
360         {
361             return;
362         }
363         for (var i = 0; i < _transitions.Count; i++)
364         {
365             var transition = _transitions.Dequeue();
366             _log.Write(transition);
367             _lastCommittedTransition = transition;
368         }
369     }
370     [MethodImpl(MethodImplOptions.AggressiveInlining)]
371     private void TransitionsPusher()
372     {
373         while (!Disposable.IsDisposed && _transitionsPusher != null)
374         {
375             Thread.Sleep(DefaultPushDelay);
376             PushTransitions();
377         }
378     }
379     [MethodImpl(MethodImplOptions.AggressiveInlining)]
380     public Transaction BeginTransaction() => new Transaction(this);
381     [MethodImpl(MethodImplOptions.AggressiveInlining)]
382     private void DisposeTransitions()
383     {
384         try
385         {
386

```

```

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 }

```

1.108 ./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 }

```

1.109 ./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>,
    ↳ IConverter<string, TLink>
11 {
12     private readonly IConverter<string, IList<TLink>> _stringToUnicodeSymbolListConverter;
13     private readonly ISequenceIndex<TLink> _index;
14     private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
15     private readonly TLink _unicodeSequenceMarker;
16
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
    ↳ IList<TLink>> stringToUnicodeSymbolListConverter, ISequenceIndex<TLink> index,
    ↳ IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
    ↳ unicodeSequenceMarker) : base(links)
19     {
20         _stringToUnicodeSymbolListConverter = stringToUnicodeSymbolListConverter;
21         _index = index;
22         _listToSequenceLinkConverter = listToSequenceLinkConverter;
23         _unicodeSequenceMarker = unicodeSequenceMarker;
24     }
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
    ↳ charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
    ↳ TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker)
28     : this(links, new
    ↳ StringToUnicodeSymbolsListConverter<TLink>(charToUnicodeSymbolConverter), index,
    ↳ listToSequenceLinkConverter, unicodeSequenceMarker) { }
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
    ↳ charToUnicodeSymbolConverter, IConverter<IList<TLink>, TLink>
    ↳ listToSequenceLinkConverter, TLink unicodeSequenceMarker)
32     : this(links, charToUnicodeSymbolConverter, new Unindex<TLink>(),
    ↳ listToSequenceLinkConverter, unicodeSequenceMarker) { }
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
    ↳ IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
    ↳ listToSequenceLinkConverter, TLink unicodeSequenceMarker)
36     : this(links, stringToUnicodeSymbolListConverter, new Unindex<TLink>(),
    ↳ listToSequenceLinkConverter, unicodeSequenceMarker) { }
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     public TLink Convert(string source)
40     {
41         var elements = _stringToUnicodeSymbolListConverter.Convert(source);
42         _index.Add(elements);
43         var sequence = _listToSequenceLinkConverter.Convert(elements);
44         return _links.GetOrCreate(sequence, _unicodeSequenceMarker);
45     }
46 }
47 }

```

1.110 ./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 }

```

1.111 ./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;

```



```

73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 public static string FromLinksToString(IList<ulong> linksList)
75 {
76     var sb = new StringBuilder();
77     for (int i = 0; i < linksList.Count; i++)
78     {
79         sb.Append(FromLinkToChar(linksList[i]));
80     }
81     return sb.ToString();
82 }
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
86 {
87     var sb = new StringBuilder();
88     if (links.Exists(link))
89     {
90         StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
91             x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
92             element =>
93             {
94                 sb.Append(FromLinkToChar(element));
95                 return true;
96             });
97     }
98     return sb.ToString();
99 }
100
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
103     ↪ chars.Length);
104
105 [MethodImpl(MethodImplOptions.AggressiveInlining)]
106 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
107 {
108     // char array to ulong array
109     var linksSequence = new ulong[count];
110     for (var i = 0; i < count; i++)
111     {
112         linksSequence[i] = FromCharToLink(chars[i]);
113     }
114     return linksSequence;
115 }
116
117 [MethodImpl(MethodImplOptions.AggressiveInlining)]
118 public static ulong[] FromStringToLinkArray(string sequence)
119 {
120     // char array to ulong array
121     var linksSequence = new ulong[sequence.Length];
122     for (var i = 0; i < sequence.Length; i++)
123     {
124         linksSequence[i] = FromCharToLink(sequence[i]);
125     }
126     return linksSequence;
127 }
128
129 [MethodImpl(MethodImplOptions.AggressiveInlining)]
130 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
131 {
132     var result = new List<ulong[]>();
133     var offset = 0;
134     while (offset < sequence.Length)
135     {
136         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
137         var relativeLength = 1;
138         var absoluteLength = offset + relativeLength;
139         while (absoluteLength < sequence.Length &&
140             ↪ currentCategory == CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
141         {
142             relativeLength++;
143             absoluteLength++;
144         }
145         // char array to ulong array
146         var innerSequence = new ulong[relativeLength];
147         var maxLength = offset + relativeLength;
148         for (var i = offset; i < maxLength; i++)
149         {

```

```

149         innerSequence[i - offset] = FromCharToLink(sequence[i]);
150     }
151     result.Add(innerSequence);
152     offset += relativeLength;
153 }
154 return result;
155 }
156
157 [MethodImpl(MethodImplOptions.AggressiveInlining)]
158 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
159 {
160     var result = new List<ulong[]>();
161     var offset = 0;
162     while (offset < array.Length)
163     {
164         var relativeLength = 1;
165         if (array[offset] <= LastCharLink)
166         {
167             var currentCategory =
168                 ↪ CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
169             var absoluteLength = offset + relativeLength;
170             while (absoluteLength < array.Length &&
171                 array[absoluteLength] <= LastCharLink &&
172                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
173                 ↪ array[absoluteLength])))
174             {
175                 relativeLength++;
176                 absoluteLength++;
177             }
178         }
179         else
180         {
181             var absoluteLength = offset + relativeLength;
182             while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
183             {
184                 relativeLength++;
185                 absoluteLength++;
186             }
187             // copy array
188             var innerSequence = new ulong[relativeLength];
189             var maxLength = offset + relativeLength;
190             for (var i = offset; i < maxLength; i++)
191             {
192                 innerSequence[i - offset] = array[i];
193             }
194             result.Add(innerSequence);
195             offset += relativeLength;
196         }
197     }
198     return result;
199 }

```

1.112 ./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         }
27     }
28 }

```

```

23     _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
24 }
25
26 [MethodImpl(MethodImplOptions.AggressiveInlining)]
27 public string Convert(TLink source)
28 {
29     if (!_unicodeSequenceCriterionMatcher.IsMatched(source))
30     {
31         throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
        ↳ not a unicode sequence.");
32     }
33     var sequence = _links.GetSource(source);
34     var charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.
        ↳ Convert).ToArray();
35     return new string(charArray);
36 }
37 }
38 }

```

1.113 ./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 }

```

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

```

1  using System;
2  using Xunit;
3  using Platform.Reflection;
4  using Platform.Memory;
5  using Platform.Scopes;
6  using Platform.Data.Doublets.Memory.United.Generic;
7
8  namespace Platform.Data.Doublets.Tests
9  {
10     public unsafe static class GenericLinksTests
11     {
12         [Fact]
13         public static void CRUDTest()
14         {
15             Using<byte>(links => links.TestCRUDOperations());
16             Using<ushort>(links => links.TestCRUDOperations());
17             Using<uint>(links => links.TestCRUDOperations());

```

```

18         Using<ulong>(links => links.TestCRUDOperations());
19     }
20
21     [Fact]
22     public static void RawNumbersCRUDTest()
23     {
24         Using<byte>(links => links.TestRawNumbersCRUDOperations());
25         Using<ushort>(links => links.TestRawNumbersCRUDOperations());
26         Using<uint>(links => links.TestRawNumbersCRUDOperations());
27         Using<ulong>(links => links.TestRawNumbersCRUDOperations());
28     }
29
30     [Fact]
31     public static void MultipleRandomCreationsAndDeletionsTest()
32     {
33         Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
34             ↪ MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
35             ↪ implementation of tree cuts out 5 bits from the address space.
36         Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
37             ↪ stMultipleRandomCreationsAndDeletions(100));
38         Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
39             ↪ MultipleRandomCreationsAndDeletions(100));
40         Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
41             ↪ tMultipleRandomCreationsAndDeletions(100));
42     }
43
44     private static void Using<TLink>(Action<ILinks<TLink>> action)
45     {
46         using (var scope = new Scope<Types<HeapResizableDirectMemory,
47             ↪ UnitedMemoryLinks<TLink>>>())
48         {
49             action(scope.Use<ILinks<TLink>>());
50         }
51     }
52 }

```

1.115 ./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 }

```

1.116 ./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     }
21 }

```

```
19     }
20 }
```

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

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

[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         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
136             ↪ totalSequenceSymbolFrequencyCounter);
137
138         var index = new
139             ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
140         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
141             ↪ ncyNumberConverter<ulong>(linkFrequenciesCache);
142         var sequenceToItsLocalElementLevelsConverter = new
143             ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
144             ↪ linkToItsFrequencyNumberConverter);
145         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
146             ↪ sequenceToItsLocalElementLevelsConverter);
147
148         var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
149             ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
150
151         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
152             ↪ index, optimalVariantConverter);
153     }
154 }
155

```

```

136 private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
137     ↳ SequenceToItsLocalElementLevelsConverter<ulong>
138     ↳ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
139     ↳ OptimalVariantConverter<ulong> optimalVariantConverter)
140 {
141     index.Add(sequence);
142
143     var optimalVariant = optimalVariantConverter.Convert(sequence);
144
145     var readSequence1 = sequences.ToList(optimalVariant);
146
147     Assert.True(sequence.SequenceEqual(readSequence1));
148 }
149
150 [Fact]
151 public static void SavedSequencesOptimizationTest()
152 {
153     LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
154     ↳ (long.MaxValue + 1UL, ulong.MaxValue));
155
156     using (var memory = new HeapResizableDirectMemory())
157     using (var disposableLinks = new UInt64UnitedMemoryLinks(memory,
158     ↳ UInt64UnitedMemoryLinks.DefaultLinksSizeStep, constants, IndexTreeType.Default))
159     {
160         var links = new UInt64Links(disposableLinks);
161
162         var root = links.CreatePoint();
163
164         //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
165         var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
166
167         var unicodeSymbolMarker = links.GetOrCreate(root,
168     ↳ addressToNumberConverter.Convert(1));
169         var unicodeSequenceMarker = links.GetOrCreate(root,
170     ↳ addressToNumberConverter.Convert(2));
171
172         var totalSequenceSymbolFrequencyCounter = new
173     ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links);
174         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
175     ↳ totalSequenceSymbolFrequencyCounter);
176         var index = new
177     ↳ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
178         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
179     ↳ ncyNumberConverter<ulong>(linkFrequenciesCache);
180         var sequenceToItsLocalElementLevelsConverter = new
181     ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
182     ↳ linkToItsFrequencyNumberConverter);
183         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
184     ↳ sequenceToItsLocalElementLevelsConverter);
185
186         var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
187     ↳ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
188
189         var unicodeSequencesOptions = new SequencesOptions<ulong>()
190     {
191             UseSequenceMarker = true,
192             SequenceMarkerLink = unicodeSequenceMarker,
193             UseIndex = true,
194             Index = index,
195             LinksToSequenceConverter = optimalVariantConverter,
196             Walker = walker,
197             UseGarbageCollection = true
198         };
199
200         var unicodeSequences = new Sequences.Sequences(new
201     ↳ SynchronizedLinks<ulong>(links), unicodeSequencesOptions);
202
203         // Create some sequences
204         var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
205     ↳ StringSplitOptions.RemoveEmptyEntries);
206         var arrays = strings.Select(x => x.Select(y =>
207     ↳ addressToNumberConverter.Convert(y)).ToArray()).ToArray();
208         for (int i = 0; i < arrays.Length; i++)
209         {
210             unicodeSequences.Create(arrays[i].ShiftRight());
211         }
212     }
213 }

```

```

196         var linksCountAfterCreation = links.Count();
197
198         // get list of sequences links
199         // for each sequence link
200         //     create new sequence version
201         //     if new sequence is not the same as sequence link
202         //         delete sequence link
203         //         collect garbadge
204         unicodeSequences.CompactAll();
205
206         var linksCountAfterCompactification = links.Count();
207
208         Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
209     }
210 }
211 }
212 }

```

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

```

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

```



```

60     }
61 }
62 }
63 }

```

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

```

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

```

1.120 ./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     }

```

1.121 ./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 =
    ↳ Default<LinksConstants<ulong>>.Instance;
22
23 static SequencesTests()
24 {
25     // Trigger static constructor to not mess with perfomance measurements
26     _ = BitString.GetBitMaskFromIndex(1);
27 }
28
29 [Fact]
30 public static void CreateAllVariantsTest()
31 {
32     const long sequenceLength = 8;
33
34     using (var scope = new TempLinksTestScope(useSequences: true))
35     {
36         var links = scope.Links;
37         var sequences = scope.Sequences;
38
39         var sequence = new ulong[sequenceLength];
40         for (var i = 0; i < sequenceLength; i++)
41         {
42             sequence[i] = links.Create();
43         }
44
45         var sw1 = Stopwatch.StartNew();
46         var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
47
48         var sw2 = Stopwatch.StartNew();
49         var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
50
51         Assert.True(results1.Count > results2.Length);
52         Assert.True(sw1.Elapsed > sw2.Elapsed);
53
54         for (var i = 0; i < sequenceLength; i++)
55         {
56             links.Delete(sequence[i]);
57         }
58
59         Assert.True(links.Count() == 0);
60     }
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,
    ↳ DefaultLinksSizeStep))
73 //    using (var links = new Links(memoryAdapter))
74 //    {
75 //        var sequence = new ulong[sequenceLength];
76 //        for (var i = 0; i < sequenceLength; i++)
77 //            sequence[i] = links.Create(itself, itself);
78 //
79 //        SequencesOptions o = new SequencesOptions();
80 //
81 //        TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
82 //        o.
83 //
84 //        var sequences = new Sequences(links);
85 //
86 //        var sw1 = Stopwatch.StartNew();
87 //        var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
88 //
89 //        var sw2 = Stopwatch.StartNew();
90 //        var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
91 //
92 //        Assert.True(results1.Count > results2.Length);
93 //        Assert.True(sw1.Elapsed > sw2.Elapsed);
94 //
95 //        for (var i = 0; i < sequenceLength; i++)
96 //            links.Delete(sequence[i]);
97 //    }
98

```

```

99 // File.Delete(tempFilename);
100 //}
101
102 [Fact]
103 public static void AllVariantsSearchTest()
104 {
105     const long sequenceLength = 8;
106
107     using (var scope = new TempLinksTestScope(useSequences: true))
108     {
109         var links = scope.Links;
110         var sequences = scope.Sequences;
111
112         var sequence = new ulong[sequenceLength];
113         for (var i = 0; i < sequenceLength; i++)
114         {
115             sequence[i] = links.Create();
116         }
117
118         var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
119
120         //for (int i = 0; i < createResults.Length; i++)
121         //    sequences.Create(createResults[i]);
122
123         var sw0 = Stopwatch.StartNew();
124         var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
125
126         var sw1 = Stopwatch.StartNew();
127         var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
128
129         var sw2 = Stopwatch.StartNew();
130         var searchResults2 = sequences.Each1(sequence); sw2.Stop();
131
132         var sw3 = Stopwatch.StartNew();
133         var searchResults3 = sequences.Each(sequence.ShiftRight()); sw3.Stop();
134
135         var intersection0 = createResults.Intersect(searchResults0).ToList();
136         Assert.True(intersection0.Count == searchResults0.Count);
137         Assert.True(intersection0.Count == createResults.Length);
138
139         var intersection1 = createResults.Intersect(searchResults1).ToList();
140         Assert.True(intersection1.Count == searchResults1.Count);
141         Assert.True(intersection1.Count == createResults.Length);
142
143         var intersection2 = createResults.Intersect(searchResults2).ToList();
144         Assert.True(intersection2.Count == searchResults2.Count);
145         Assert.True(intersection2.Count == createResults.Length);
146
147         var intersection3 = createResults.Intersect(searchResults3).ToList();
148         Assert.True(intersection3.Count == searchResults3.Count);
149         Assert.True(intersection3.Count == createResults.Length);
150
151         for (var i = 0; i < sequenceLength; i++)
152         {
153             links.Delete(sequence[i]);
154         }
155     }
156 }
157
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();
243
244         //Global.Trash = searchResults3;
245
246         //var searchResults1Strings = searchResults1.Select(x => x + ": " +
247             ↪ sequences.FormatSequence(x)).ToList();
248         //Global.Trash = searchResults1Strings;
249
250         var intersection1 = createResults.Intersect(searchResults1).ToList();
251         Assert.True(intersection1.Count == createResults.Length);
252
253         var intersection2 = createResults.Intersect(searchResults2).ToList();
254         Assert.True(intersection2.Count == createResults.Length);
255
256         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
278         var balancedVariant = balancedVariantConverter.Convert(sequence);
279
280         var partialSequence = new ulong[sequenceLength - 2];
281
282         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
283
284         var sw1 = Stopwatch.StartNew();
285         var searchResults1 =
286             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
287
288         var sw2 = Stopwatch.StartNew();
289         var searchResults2 =
290             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
291
292         Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
293
294         Assert.True(searchResults2.Count == 1 && balancedVariant ==
295             ↪ searchResults2.First());
296
297         for (var i = 0; i < sequenceLength; i++)
298         {
299             links.Delete(sequence[i]);
300         }
301     }
302 }
303
304 [Fact(Skip = "Correct implementation is pending")]
305 public static void PatternMatchTest()
306 {
307     var zeroOrMany = Sequences.Sequences.ZeroOrMany;
308
309     using (var scope = new TempLinksTestScope(useSequences: true))
310     {
311         var links = scope.Links;
312         var sequences = scope.Sequences;
313
314         var e1 = links.Create();
315         var e2 = links.Create();
316
317         var sequence = new[]
318         {
319             e1, e2, e1, e2 // mama / papa
320         };
321
322         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
323
324         var balancedVariant = balancedVariantConverter.Convert(sequence);
325
326         // 1: [1]
327         // 2: [2]
328         // 3: [1,2]
329         // 4: [1,2,1,2]

```

```

328     var doublet = links.GetSource(balancedVariant);
329
330     var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
331
332     Assert.True(matchedSequences1.Count == 0);
333
334     var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
335
336     Assert.True(matchedSequences2.Count == 0);
337
338     var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
339
340     Assert.True(matchedSequences3.Count == 0);
341
342     var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
343
344     Assert.Contains(doublet, matchedSequences4);
345     Assert.Contains(balancedVariant, matchedSequences4);
346
347     for (var i = 0; i < sequence.Length; i++)
348     {
349         links.Delete(sequence[i]);
350     }
351 }
352 }
353
354 [Fact]
355 public static void IndexTest()
356 {
357     using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
↪ 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/%
↪ D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87
↪ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
380 private static readonly string _exampleText =
381     @"([english
↪ version] (https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
382
383

```

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

[[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>)

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?

[[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>)

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

392
393 `[[белая точка, чёрная`
↪ `точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png "белая`
↪ `точка, чёрная`
↪ `точка")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png)`
394
395 `А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет`
↪ `если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой?`
↪ `Гранью? Разделителем? Единицей?`
396
397 `[[две белые точки, чёрная вертикальная`
↪ `линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png "две`
↪ `белые точки, чёрная вертикальная`
↪ `линия")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)`
398
399 `Можно ли замкнуть движение? Может ли это быть кругом? Можно ли замкнуть время? Или остаётся`
↪ `только спираль? Но что если замкнуть предел? Создать ограничение, разделение? Получится`
↪ `замкнутая область? Полностью отделённая от всего остального? Но что это всё остальное? Что`
↪ `можно делить? В каком направлении? Ничего или всё? Пустота или полнота? Начало или конец?`
↪ `Или может быть это единица и ноль? Дуальность? Противоположность? А что будет с кругом если`
↪ `у него нет размера? Будет ли круг точкой? Точка состоящая из точек?`
400
401 `[[белая вертикальная линия, чёрный`
↪ `круг](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png "белая`
↪ `вертикальная линия, чёрный`
↪ `круг")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png)`
402
403 `Как ещё можно использовать грань, черту, линию? А что если она может что-то соединять, может`
↪ `тогда её нужно повернуть? Почему то, что перпендикулярно вертикальному горизонтально?`
↪ `Горизонт? Инвертирует ли это смысл? Что такое смысл? Из чего состоит смысл? Существует ли`
↪ `элементарная единица смысла?`
404
405 `[[белый круг, чёрная горизонтальная`
↪ `линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png "белый`
↪ `круг, чёрная горизонтальная`
↪ `линия")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png)`
406
407 `Соединять, допустим, а какой смысл в этом есть ещё? Что если помимо смысла "соединить,`
↪ `связать", есть ещё и смысл направления "от начала к концу"? От предка к потомку? От`
↪ `родителя к ребёнку? От общего к частному?`
408
409 `[[белая горизонтальная линия, чёрная горизонтальная`
↪ `стрелка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png`
↪ `"белая горизонтальная линия, чёрная горизонтальная`
↪ `стрелка")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png)`
410
411 `Шаг назад. Возьмём опять отделённую область, которая лишь та же замкнутая линия, что ещё она`
↪ `может представлять собой? Объект? Но в чём его суть? Разве не в том, что у него есть`
↪ `граница, разделяющая внутреннее и внешнее? Допустим связь, стрелка, линия соединяет два`
↪ `объекта, как бы это выглядело?`
412
413 `[[белая связь, чёрная направленная`
↪ `связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png "белая`
↪ `связь, чёрная направленная`
↪ `связь")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png)`
414
415 `Допустим у нас есть смысл "связать" и смысл "направления", много ли это нам даёт? Много ли`
↪ `вариантов интерпретации? А что если уточнить, каким именно образом выполнена связь? Что если`
↪ `можно задать ей чёткий, конкретный смысл? Что это будет? Тип? Глагол? Связка? Действие?`
↪ `Трансформация? Переход из состояния в состояние? Или всё это и есть объект, суть которого в`
↪ `его конечном состоянии, если конечно конец определён направлением?`
416
417 `[[белая обычная и направленная связи, чёрная типизированная`
↪ `связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png "белая`
↪ `обычная и направленная связи, чёрная типизированная`
↪ `связь")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png)`
418
419 `А что если всё это время, мы смотрели на суть как бы снаружи? Можно ли взглянуть на это изнутри?`
↪ `Что будет внутри объектов? Объекты ли это? Или это связи? Может ли эта структура описать`
↪ `сама себя? Но что тогда получится, разве это не рекурсия? Может это фрактал?`
420
421 `[[белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная`
↪ `связь с рекурсивной внутренней`
↪ `структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png`
↪ `"белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная`
↪ `типизированная связь с рекурсивной внутренней структурой")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png)`
422
423 `На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом`
↪ `рекурсии или фрактала?`


```

424
425 [![белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с двойной рекурсивной внутренней
    ↳ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png
    ↳ ""белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
    ↳ типизированная связь с двойной рекурсивной внутренней структурой"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)
426
427 Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
    ↳ Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
428
429 [![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности,
    ↳ чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://
    ↳ /raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png ""белая обычная и
    ↳ направленная связи со структурой из 8 цветных элементов последовательности, чёрная
    ↳ типизированная связь со структурой из 8 цветных элементов последовательности"")](https://raw
    ↳ .githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)
430
431 ...
432
433 [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-anim
    ↳ ation-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(links.Unsync);
456         var totalSequenceSymbolFrequencyCounter = new
            ↳ TotalSequenceSymbolFrequencyCounter(links.Unsync);
457         var doubletFrequenciesCache = new LinkFrequenciesCache(links.Unsync,
            ↳ totalSequenceSymbolFrequencyCounter);
458         var compressingConverter = new CompressingConverter(links.Unsync,
            ↳ balancedVariantConverter, doubletFrequenciesCache);
459
460         var compressedVariant = compressingConverter.Convert(sequence);
461
462         // 1: [1]          (1->1) point
463         // 2: [2]          (2->2) point
464         // 3: [1,2]        (1->2) doublet
465         // 4: [1,2,1,2]    (3->3) doublet
466
467         Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
468         Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
469         Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
470         Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
471
472         var source = _constants.SourcePart;
473         var target = _constants.TargetPart;
474
475         Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
476         Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
477         Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
478         Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
479
480         // 4 - length of sequence
481         Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
            ↳ == sequence[0]);

```

```

482     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
483         ↪ == sequence[1]);
484     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
485         ↪ == sequence[2]);
486     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
487         ↪ == sequence[3]);
488 }
489 [Fact]
490 public static void CompressionEfficiencyTest()
491 {
492     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
493         ↪ StringSplitOptions.RemoveEmptyEntries);
494     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
495     var totalCharacters = arrays.Select(x => x.Length).Sum();
496
497     using (var scope1 = new TempLinksTestScope(useSequences: true))
498     using (var scope2 = new TempLinksTestScope(useSequences: true))
499     using (var scope3 = new TempLinksTestScope(useSequences: true))
500     {
501         scope1.Links.Unsync.UseUnicode();
502         scope2.Links.Unsync.UseUnicode();
503         scope3.Links.Unsync.UseUnicode();
504
505         var balancedVariantConverter1 = new
506             ↪ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
507         var totalSequenceSymbolFrequencyCounter = new
508             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
509         var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
510             ↪ totalSequenceSymbolFrequencyCounter);
511         var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
512             ↪ balancedVariantConverter1, linkFrequenciesCache1,
513             ↪ doInitialFrequenciesIncrement: false);
514
515         //var compressor2 = scope2.Sequences;
516         var compressor3 = scope3.Sequences;
517
518         var constants = Default<LinksConstants<ulong>>.Instance;
519
520         var sequences = compressor3;
521         //var meaningRoot = links.CreatePoint();
522         //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
523         //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
524         //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
525             ↪ constants.Itself);
526
527         //var unaryNumberToAddressConverter = new
528             ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
529         //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
530             ↪ unaryOne);
531         //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
532             ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
533         //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
534             ↪ frequencyPropertyMarker, frequencyMarker);
535         //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
536             ↪ frequencyPropertyOperator, frequencyIncrementer);
537         //var linkToItsFrequencyNumberConverter = new
538             ↪ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
539             ↪ unaryNumberToAddressConverter);
540
541         var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
542             ↪ totalSequenceSymbolFrequencyCounter);
543
544         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
545             ↪ ncyNumberConverter<ulong>(linkFrequenciesCache3);
546
547         var sequenceToItsLocalElementLevelsConverter = new
548             ↪ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
549             ↪ linkToItsFrequencyNumberConverter);
550         var optimalVariantConverter = new
551             ↪ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
552             ↪ sequenceToItsLocalElementLevelsConverter);
553
554         var compressed1 = new ulong[arrays.Length];
555         var compressed2 = new ulong[arrays.Length];
556         var compressed3 = new ulong[arrays.Length];

```

```

536
537 var START = 0;
538 var END = arrays.Length;
539
540 //for (int i = START; i < END; i++)
541 //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
542
543 var initialCount1 = scope2.Links.Unsync.Count();
544
545 var sw1 = Stopwatch.StartNew();
546
547 for (int i = START; i < END; i++)
548 {
549     linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
550     compressed1[i] = compressor1.Convert(arrays[i]);
551 }
552
553 var elapsed1 = sw1.Elapsed;
554
555 var balancedVariantConverter2 = new
556     ↪ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
557
558 var initialCount2 = scope2.Links.Unsync.Count();
559
560 var sw2 = Stopwatch.StartNew();
561
562 for (int i = START; i < END; i++)
563 {
564     compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
565 }
566
567 var elapsed2 = sw2.Elapsed;
568
569 for (int i = START; i < END; i++)
570 {
571     linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
572 }
573
574 var initialCount3 = scope3.Links.Unsync.Count();
575
576 var sw3 = Stopwatch.StartNew();
577
578 for (int i = START; i < END; i++)
579 {
580     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
581     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
582 }
583
584 var elapsed3 = sw3.Elapsed;
585
586 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
587     ↪ Optimal variant: {elapsed3}");
588
589 // Assert.True(elapsed1 > elapsed2);
590
591 // Checks
592 for (int i = START; i < END; i++)
593 {
594     var sequence1 = compressed1[i];
595     var sequence2 = compressed2[i];
596     var sequence3 = compressed3[i];
597
598     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
599         ↪ scope1.Links.Unsync);
600
601     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
602         ↪ scope2.Links.Unsync);
603
604     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
605         ↪ scope3.Links.Unsync);
606
607     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
608         ↪ link.IsPartialPoint());
609     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
610         ↪ link.IsPartialPoint());
611     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
612         ↪ link.IsPartialPoint());

```

```

606         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
        ↪     arrays[i].Length > 3)
607         //    Assert.False(structure1 == structure2);
608         //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
        ↪     arrays[i].Length > 3)
        //    Assert.False(structure3 == structure2);
609
610         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
611         Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
612     }
613
614     Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
        ↪     totalCharacters);
615     Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
        ↪     totalCharacters);
616     Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
        ↪     totalCharacters);
617
618     Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
        ↪     totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
        ↪     totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
        ↪     totalCharacters}");
619
620     Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
        ↪     scope2.Links.Unsync.Count() - initialCount2);
621     Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
        ↪     scope2.Links.Unsync.Count() - initialCount2);
622
623     var duplicateProvider1 = new
        ↪     DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
624     var duplicateProvider2 = new
        ↪     DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
625     var duplicateProvider3 = new
        ↪     DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
626
627     var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
628     var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
629     var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
630
631     var duplicates1 = duplicateCounter1.Count();
632
633     ConsoleHelpers.Debug("-----");
634
635     var duplicates2 = duplicateCounter2.Count();
636
637     ConsoleHelpers.Debug("-----");
638
639     var duplicates3 = duplicateCounter3.Count();
640
641     Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
642
643     linkFrequenciesCache1.ValidateFrequencies();
644     linkFrequenciesCache3.ValidateFrequencies();
645 }
646
647 [Fact]
648 public static void CompressionStabilityTest()
649 {
650     // TODO: Fix bug (do a separate test)
651     //const ulong minNumbers = 0;
652     //const ulong maxNumbers = 1000;
653
654     const ulong minNumbers = 10000;
655     const ulong maxNumbers = 12500;
656
657     var strings = new List<string>();
658
659     for (ulong i = minNumbers; i < maxNumbers; i++)
660     {
661         strings.Add(i.ToString());
662     }
663
664     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
665     var totalCharacters = arrays.Select(x => x.Length).Sum();
666
667
668

```

```

669 using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
    ↳ SequencesOptions<ulong> { UseCompression = true,
    ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
670 using (var scope2 = new TempLinksTestScope(useSequences: true))
671 {
672     scope1.Links.UseUnicode();
673     scope2.Links.UseUnicode();
674
675     //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
676     var compressor1 = scope1.Sequences;
677     var compressor2 = scope2.Sequences;
678
679     var compressed1 = new ulong[arrays.Length];
680     var compressed2 = new ulong[arrays.Length];
681
682     var sw1 = Stopwatch.StartNew();
683
684     var START = 0;
685     var END = arrays.Length;
686
687     // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
688     // Stability issue starts at 10001 or 11000
689     //for (int i = START; i < END; i++)
690     //{
691     //    var first = compressor1.Compress(arrays[i]);
692     //    var second = compressor1.Compress(arrays[i]);
693
694     //    if (first == second)
695     //        compressed1[i] = first;
696     //    else
697     //    {
698     //        // TODO: Find a solution for this case
699     //    }
700     //}
701
702     for (int i = START; i < END; i++)
703     {
704         var first = compressor1.Create(arrays[i].ShiftRight());
705         var second = compressor1.Create(arrays[i].ShiftRight());
706
707         if (first == second)
708         {
709             compressed1[i] = first;
710         }
711         else
712         {
713             // TODO: Find a solution for this case
714         }
715     }
716
717     var elapsed1 = sw1.Elapsed;
718
719     var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
720
721     var sw2 = Stopwatch.StartNew();
722
723     for (int i = START; i < END; i++)
724     {
725         var first = balancedVariantConverter.Convert(arrays[i]);
726         var second = balancedVariantConverter.Convert(arrays[i]);
727
728         if (first == second)
729         {
730             compressed2[i] = first;
731         }
732     }
733
734     var elapsed2 = sw2.Elapsed;
735
736     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
    ↳ {elapsed2}");
737
738     Assert.True(elapsed1 > elapsed2);
739
740     // Checks
741     for (int i = START; i < END; i++)
742     {
743         var sequence1 = compressed1[i];
744         var sequence2 = compressed2[i];

```

```

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

```

```

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

```

```

888         for (var i = 0; i < sequenceLength; i++)
889         {
890             links.Delete(sequence[i]);
891         }
892     }
893 }
894
895 [Fact]
896 public static void AllPossibleConnectionsTest()
897 {
898     const long sequenceLength = 5;
899
900     using (var scope = new TempLinksTestScope(useSequences: true))
901     {
902         var links = scope.Links;
903         var sequences = scope.Sequences;
904
905         var sequence = new ulong[sequenceLength];
906         for (var i = 0; i < sequenceLength; i++)
907         {
908             sequence[i] = links.Create();
909         }
910
911         var createResults = sequences.CreateAllVariants2(sequence);
912         var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
913
914         for (var i = 0; i < 1; i++)
915         {
916             var sw1 = Stopwatch.StartNew();
917             var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
918
919             var sw2 = Stopwatch.StartNew();
920             var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
921
922             var sw3 = Stopwatch.StartNew();
923             var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
924
925             var sw4 = Stopwatch.StartNew();
926             var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
927
928             Global.Trash = searchResults3;
929             Global.Trash = searchResults4; //-V3008
930
931             var intersection1 = createResults.Intersect(searchResults1).ToList();
932             Assert.True(intersection1.Count == createResults.Length);
933
934             var intersection2 = reverseResults.Intersect(searchResults1).ToList();
935             Assert.True(intersection2.Count == reverseResults.Length);
936
937             var intersection0 = searchResults1.Intersect(searchResults2).ToList();
938             Assert.True(intersection0.Count == searchResults2.Count);
939
940             var intersection3 = searchResults2.Intersect(searchResults3).ToList();
941             Assert.True(intersection3.Count == searchResults3.Count);
942
943             var intersection4 = searchResults3.Intersect(searchResults4).ToList();
944             Assert.True(intersection4.Count == searchResults4.Count);
945         }
946
947         for (var i = 0; i < sequenceLength; i++)
948         {
949             links.Delete(sequence[i]);
950         }
951     }
952 }
953
954 [Fact(Skip = "Correct implementation is pending")]
955 public static void CalculateAllUsagesTest()
956 {
957     const long sequenceLength = 3;
958
959     using (var scope = new TempLinksTestScope(useSequences: true))
960     {
961         var links = scope.Links;
962         var sequences = scope.Sequences;
963
964         var sequence = new ulong[sequenceLength];
965         for (var i = 0; i < sequenceLength; i++)
966         {
967

```



```

968         sequence[i] = links.Create();
969     }
970
971     var createResults = sequences.CreateAllVariants2(sequence);
972
973     //var reverseResults =
974     ↪ sequences.CreateAllVariants2(sequence.Reverse().ToArray());
975
976     for (var i = 0; i < 1; i++)
977     {
978         var linksTotalUsages1 = new ulong[links.Count() + 1];
979
980         sequences.CalculateAllUsages(linksTotalUsages1);
981
982         var linksTotalUsages2 = new ulong[links.Count() + 1];
983
984         sequences.CalculateAllUsages2(linksTotalUsages2);
985
986         var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
987         Assert.True(intersection1.Count == linksTotalUsages2.Length);
988     }
989
990     for (var i = 0; i < sequenceLength; i++)
991     {
992         links.Delete(sequence[i]);
993     }
994 }
995 }
996 }

```

1.122 ./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));
38             Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
39             ↪ tMultipleRandomCreationsAndDeletions(100));
40         }
41
42         private static void Using<TLink>(Action<ILinks<TLink>> action)
43         {
44             using (var dataMemory = new HeapResizableDirectMemory())
45             using (var indexMemory = new HeapResizableDirectMemory())
46             using (var memory = new SplitMemoryLinks<TLink>(dataMemory, indexMemory))
47             {

```

```

43         action(memory);
44     }
45 }
46
47 private static void UsingWithExternalReferences<TLink>(Action<ILinks<TLink>> action)
48 {
49     var contants = 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,
53         ↪ SplitMemoryLinks<TLink>.DefaultLinksSizeStep, contants))
54     {
55         action(memory);
56     }
57 }
58 }

```

1.123 ./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) { }
21
22         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
23             ↪ true, bool useSequences = false, bool useLog = false)
24         {
25             _deleteFiles = deleteFiles;
26             TempFilename = Path.GetTempFileName();
27             TempTransactionLogFilename = Path.GetTempFileName();
28             var coreMemoryAdapter = new UInt64UnitedMemoryLinks(TempFilename);
29             MemoryAdapter = useLog ? (ILinks<ulong>)new
30                 ↪ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
31                 ↪ coreMemoryAdapter;
32             Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
33             if (useSequences)
34             {
35                 Sequences = new Sequences.Sequences(Links, sequencesOptions);
36             }
37         }
38
39         protected override void Dispose(bool manual, bool wasDisposed)
40         {
41             if (!wasDisposed)
42             {
43                 Links.Unsync.DisposeIfPossible();
44                 if (_deleteFiles)
45                 {
46                     DeleteFiles();
47                 }
48             }
49         }
50
51         public void DeleteFiles()
52         {
53             File.Delete(TempFilename);
54             File.Delete(TempTransactionLogFilename);
55         }
56     }
57 }

```

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

```

1  using System.Collections.Generic;
2  using Xunit;
3  using Platform.Ranges;
4  using Platform.Numbers;
5  using Platform.Random;
6  using Platform.Setters;
7  using Platform.Converters;
8
9  namespace Platform.Data.Doublets.Tests
10 {
11     public static class TestExtensions
12     {
13         public static void TestCRUDOperations<T>(this ILinks<T> links)
14         {
15             var constants = links.Constants;
16
17             var equalityComparer = EqualityComparer<T>.Default;
18
19             var zero = default(T);
20             var one = Arithmetic.Increment(zero);
21
22             // Create Link
23             Assert.True(equalityComparer.Equals(links.Count(), zero));
24
25             var setter = new Setter<T>(constants.Null);
26             links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
27
28             Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
29
30             var linkAddress = links.Create();
31
32             var link = new Link<T>(links.GetLink(linkAddress));
33
34             Assert.True(link.Count == 3);
35             Assert.True(equalityComparer.Equals(link.Index, linkAddress));
36             Assert.True(equalityComparer.Equals(link.Source, constants.Null));
37             Assert.True(equalityComparer.Equals(link.Target, constants.Null));
38
39             Assert.True(equalityComparer.Equals(links.Count(), one));
40
41             // Get first link
42             setter = new Setter<T>(constants.Null);
43             links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
44
45             Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
46
47             // Update link to reference itself
48             links.Update(linkAddress, linkAddress, linkAddress);
49
50             link = new Link<T>(links.GetLink(linkAddress));
51
52             Assert.True(equalityComparer.Equals(link.Source, linkAddress));
53             Assert.True(equalityComparer.Equals(link.Target, linkAddress));
54
55             // Update link to reference null (prepare for delete)
56             var updated = links.Update(linkAddress, constants.Null, constants.Null);
57
58             Assert.True(equalityComparer.Equals(updated, linkAddress));
59
60             link = new Link<T>(links.GetLink(linkAddress));
61
62             Assert.True(equalityComparer.Equals(link.Source, constants.Null));
63             Assert.True(equalityComparer.Equals(link.Target, constants.Null));
64
65             // Delete link
66             links.Delete(linkAddress);
67
68             Assert.True(equalityComparer.Equals(links.Count(), zero));
69
70             setter = new Setter<T>(constants.Null);
71             links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
72
73             Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
74         }
75
76         public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
77         {
78             // Constants
79             var constants = links.Constants;
80             var equalityComparer = EqualityComparer<T>.Default;

```

```

81
82     var zero = default(T);
83     var one = Arithmetic.Increment(zero);
84     var two = Arithmetic.Increment(one);
85
86     var h106E = new Hybrid<T>(106L, isExternal: true);
87     var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
88     var h108E = new Hybrid<T>(-108L);
89
90     Assert.Equal(106L, h106E.AbsoluteValue);
91     Assert.Equal(107L, h107E.AbsoluteValue);
92     Assert.Equal(108L, h108E.AbsoluteValue);
93
94     // Create Link (External -> External)
95     var linkAddress1 = links.Create();
96
97     links.Update(linkAddress1, h106E, h108E);
98
99     var link1 = new Link<T>(links.GetLink(linkAddress1));
100
101     Assert.True(equalityComparer.Equals(link1.Source, h106E));
102     Assert.True(equalityComparer.Equals(link1.Target, h108E));
103
104     // Create Link (Internal -> External)
105     var linkAddress2 = links.Create();
106
107     links.Update(linkAddress2, linkAddress1, h108E);
108
109     var link2 = new Link<T>(links.GetLink(linkAddress2));
110
111     Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
112     Assert.True(equalityComparer.Equals(link2.Target, h108E));
113
114     // Create Link (Internal -> Internal)
115     var linkAddress3 = links.Create();
116
117     links.Update(linkAddress3, linkAddress1, linkAddress2);
118
119     var link3 = new Link<T>(links.GetLink(linkAddress3));
120
121     Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
122     Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
123
124     // Search for created link
125     var setter1 = new Setter<T>(constants.Null);
126     links.Each(h106E, h108E, setter1.SetAndReturnFalse);
127
128     Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
129
130     // Search for nonexistent link
131     var setter2 = new Setter<T>(constants.Null);
132     links.Each(h106E, h107E, setter2.SetAndReturnFalse);
133
134     Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
135
136     // Update link to reference null (prepare for delete)
137     var updated = links.Update(linkAddress3, constants.Null, constants.Null);
138
139     Assert.True(equalityComparer.Equals(updated, linkAddress3));
140
141     link3 = new Link<T>(links.GetLink(linkAddress3));
142
143     Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
144     Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
145
146     // Delete link
147     links.Delete(linkAddress3);
148
149     Assert.True(equalityComparer.Equals(links.Count(), two));
150
151     var setter3 = new Setter<T>(constants.Null);
152     links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
153
154     Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
155 }
156
157 public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
    ↪ links, int maximumOperationsPerCycle)
158 {
159     var comparer = Comparer<TLink>.Default;

```

```

160     var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
161     var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
162     for (var N = 1; N < maximumOperationsPerCycle; N++)
163     {
164         var random = new System.Random(N);
165         var created = 0UL;
166         var deleted = 0UL;
167         for (var i = 0; i < N; i++)
168         {
169             var linksCount = addressToUInt64Converter.Convert(links.Count());
170             var createPoint = random.NextBoolean();
171             if (linksCount > 2 && createPoint)
172             {
173                 var linksAddressRange = new Range<ulong>(1, linksCount);
174                 TLink source = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA_
                    ↪ ddressRange));
175                 TLink target = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA_
                    ↪ ddressRange));
                    ↪ //-V3086
176                 var resultLink = links.GetOrCreate(source, target);
177                 if (comparer.Compare(resultLink,
                    ↪ uInt64ToAddressConverter.Convert(linksCount)) > 0)
178                 {
179                     created++;
180                 }
181             }
182             else
183             {
184                 links.Create();
185                 created++;
186             }
187         }
188         Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
189         for (var i = 0; i < N; i++)
190         {
191             TLink link = uInt64ToAddressConverter.Convert((ulong)i + 1UL);
192             if (links.Exists(link))
193             {
194                 links.Delete(link);
195                 deleted++;
196             }
197         }
198         Assert.True(addressToUInt64Converter.Convert(links.Count()) == 0L);
199     }
200 }
201 }
202 }

```

1.125 ./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 =
                ↪ Default<LinksConstants<ulong>>.Instance;
28
29         private const long Iterations = 10 * 1024;
30

```

#region Concept

[Fact]

public static void MultipleCreateAndDeleteTest()

```
{
    using (var scope = new Scope<Types<HeapResizableDirectMemory,
        ↳ UInt64UnitedMemoryLinks>>())
    {
        new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeleti
            ↳ ons(100);
    }
}
```

[Fact]

public static void CascadeUpdateTest()

```
{
    var itself = _constants.Itself;
    using (var scope = new TempLinksTestScope(useLog: true))
    {
        var links = scope.Links;

        var l1 = links.Create();
        var l2 = links.Create();

        l2 = links.Update(l2, l2, l1, l2);

        links.CreateAndUpdate(l2, itself);
        links.CreateAndUpdate(l2, itself);

        l2 = links.Update(l2, l1);

        links.Delete(l2);

        Global.Trash = links.Count();

        links.Unsync.DisposeIfPossible(); // Close links to access log

        Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
            ↳ e.TempTransactionLogFilename);
    }
}
```

[Fact]

public static void BasicTransactionLogTest()

```
{
    using (var scope = new TempLinksTestScope(useLog: true))
    {
        var links = scope.Links;
        var l1 = links.Create();
        var l2 = links.Create();

        Global.Trash = links.Update(l2, l2, l1, l2);

        links.Delete(l1);

        links.Unsync.DisposeIfPossible(); // Close links to access log

        Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
            ↳ e.TempTransactionLogFilename);
    }
}
```

[Fact]

public static void TransactionAutoRevertedTest()

```
{
    // Auto Reverted (Because no commit at transaction)
    using (var scope = new TempLinksTestScope(useLog: true))
    {
        var links = scope.Links;
        var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
        using (var transaction = transactionsLayer.BeginTransaction())
        {
            var l1 = links.Create();
            var l2 = links.Create();

            links.Update(l2, l2, l1, l2);
        }

        Assert.Equal(0UL, links.Count());
    }
}
```

```

links.Unsync.DisposeIfPossible();

var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
Assert.Single(transitions);
}
}

[Fact]
public static void TransactionUserCodeErrorNoDataSavedTest()
{
    // User Code Error (Autoreverted), no data saved
    var itself = _constants.Itself;

    TempLinksTestScope lastScope = null;
    try
    {
        using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
            useLog: true))
        {
            var links = scope.Links;
            var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecoratorBase<ulong>)links.Unsync).Links;
            using (var transaction = transactionsLayer.BeginTransaction())
            {
                var l1 = links.CreateAndUpdate(itself, itself);
                var l2 = links.CreateAndUpdate(itself, itself);

                l2 = links.Update(l2, l2, l1, l2);

                links.CreateAndUpdate(l2, itself);
                links.CreateAndUpdate(l2, itself);

                //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);

                l2 = links.Update(l2, l1);

                links.Delete(l2);

                ExceptionThrower();

                transaction.Commit();
            }

            Global.Trash = links.Count();
        }
    }
    catch
    {
        Assert.False(lastScope == null);

        var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(lastScope.TempTransactionLogFilename);

        Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
            transitions[0].After.IsNull());

        lastScope.DeleteFiles();
    }
}

[Fact]
public static void TransactionUserCodeErrorSomeDataSavedTest()
{
    // User Code Error (Autoreverted), some data saved
    var itself = _constants.Itself;

    TempLinksTestScope lastScope = null;
    try
    {
        ulong l1;
        ulong l2;

        using (var scope = new TempLinksTestScope(useLog: true))
        {
            var links = scope.Links;
            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,
192         ↪ useLog: true))
193     {
194         var links = scope.Links;
195         var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
196         using (var transaction = transactionsLayer.BeginTransaction())
197         {
198             l2 = links.Update(l2, l1);
199
200             links.Delete(l2);
201
202             ExceptionThrower();
203
204             transaction.Commit();
205         }
206
207         Global.Trash = links.Count();
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
219 [Fact]
220 public static void TransactionCommit()
221 {
222     var itself = _constants.Itself;
223
224     var tempDatabaseFilename = Path.GetTempFileName();
225     var tempTransactionLogFilename = Path.GetTempFileName();
226
227     // Commit
228     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
            ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
229     using (var links = new UInt64Links(memoryAdapter))
230     {
231         using (var transaction = memoryAdapter.BeginTransaction())
232         {
233             var l1 = links.CreateAndUpdate(itself, itself);
234             var l2 = links.CreateAndUpdate(itself, itself);
235
236             Global.Trash = links.Update(l2, l2, l1, l2);
237
238             links.Delete(l1);
239
240             transaction.Commit();
241         }
242
243         Global.Trash = links.Count();
244     }
245
246     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
            ↪ sactionLogFilename);
247 }
248
249 [Fact]
250 public static void TransactionDamage()
251 {
252     var itself = _constants.Itself;

```



```

253
254     var tempDatabaseFilename = Path.GetTempFileName();
255     var tempTransactionLogFilename = Path.GetTempFileName();
256
257     // Commit
258     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
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
282     FileHelpers.WriteFirst(tempTransactionLogFilename, new
283         ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
284
285     // Try load damaged database
286     try
287     {
288         // TODO: Fix
289         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
290             ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
291         using (var links = new UInt64Links(memoryAdapter))
292         {
293             Global.Trash = links.Count();
294         }
295     }
296     catch (NotSupportedException ex)
297     {
298         Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
299             ↪ yet.");
300     }
301
302     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
303         ↪ sactionLogFilename);
304
305     File.Delete(tempDatabaseFilename);
306     File.Delete(tempTransactionLogFilename);
307 }
308
309 [Fact]
310 public static void Bug1Test()
311 {
312     var tempDatabaseFilename = Path.GetTempFileName();
313     var tempTransactionLogFilename = Path.GetTempFileName();
314
315     var itself = _constants.Itself;
316
317     // User Code Error (Autoreverted), some data saved
318     try
319     {
320         ulong l1;
321         ulong l2;
322
323         using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
324         using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
325             ↪ tempTransactionLogFilename))
326         using (var links = new UInt64Links(memoryAdapter))
327         {
328             l1 = links.CreateAndUpdate(itself, itself);
329             l2 = links.CreateAndUpdate(itself, itself);
330
331             l2 = links.Update(l2, l2, l1, l2);
332         }
333     }
334 }

```

```

325         links.CreateAndUpdate(l2, itself);
326         links.CreateAndUpdate(l2, itself);
327     }
328
329     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp ↵
330         ↵ TransactionLogFilename);
331
332     using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
333     using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
334         ↵ tempTransactionLogFilename))
335     using (var links = new UInt64Links(memoryAdapter))
336     {
337         using (var transaction = memoryAdapter.BeginTransaction())
338         {
339             l2 = links.Update(l2, l1);
340
341             links.Delete(l2);
342
343             ExceptionThrower();
344
345             transaction.Commit();
346         }
347
348         Global.Trash = links.Count();
349     }
350     catch
351     {
352         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp ↵
353         ↵ TransactionLogFilename);
354     }
355
356     File.Delete(tempDatabaseFilename);
357     File.Delete(tempTransactionLogFilename);
358 }
359
360 private static void ExceptionThrower() => throw new InvalidOperationException();
361
362 [Fact]
363 public static void PathsTest()
364 {
365     var source = _constants.SourcePart;
366     var target = _constants.TargetPart;
367
368     using (var scope = new TempLinksTestScope())
369     {
370         var links = scope.Links;
371         var l1 = links.CreatePoint();
372         var l2 = links.CreatePoint();
373
374         var r1 = links.GetByKeys(l1, source, target, source);
375         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
376     }
377 }
378
379 [Fact]
380 public static void RecursiveStringFormattingTest()
381 {
382     using (var scope = new TempLinksTestScope(useSequences: true))
383     {
384         var links = scope.Links;
385         var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
386
387         var a = links.CreatePoint();
388         var b = links.CreatePoint();
389         var c = links.CreatePoint();
390
391         var ab = links.GetOrCreate(a, b);
392         var cb = links.GetOrCreate(c, b);
393         var ac = links.GetOrCreate(a, c);
394
395         a = links.Update(a, c, b);
396         b = links.Update(b, a, c);
397         c = links.Update(c, a, b);
398
399         Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
400         Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
401         Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));

```

```

401     Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
402         ↪ "(5:(4:5 (6:5 4)) 6)");
403     Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
404         ↪ "(6:(5:(4:5 6) 6) 4)");
405     Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
406         ↪ "(4:(5:4 (6:5 4)) 6)");
407
408     // TODO: Think how to build balanced syntax tree while formatting structure (eg.
409     ↪ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)")
410
411     Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
412         ↪ "{5}{5}{4}{6}");
413     Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
414         ↪ "{5}{6}{6}{4}");
415     Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
416         ↪ "{4}{5}{4}{6}");
417 }
418 }
419
420 private static void DefaultFormatter(StringBuilder sb, ulong link)
421 {
422     sb.Append(link.ToString());
423 }
424
425 #endregion
426
427 #region Performance
428
429 /*
430 public static void RunAllPerformanceTests()
431 {
432     try
433     {
434         links.TestLinksInSteps();
435     }
436     catch (Exception ex)
437     {
438         ex.WriteToConsole();
439     }
440
441     return;
442
443     try
444     {
445         //ThreadPool.SetMaxThreads(2, 2);
446
447         // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
448         ↪ результат
449         // Также это дополнительно помогает в отладке
450         // Увеличивает вероятность попадания информации в кэши
451         for (var i = 0; i < 10; i++)
452         {
453             //0 - 10 ГБ
454             //Каждые 100 МБ срез цифр
455
456             //links.TestGetSourceFunction();
457             //links.TestGetSourceFunctionInParallel();
458             //links.TestGetTargetFunction();
459             //links.TestGetTargetFunctionInParallel();
460             links.Create64BillionLinks();
461
462             links.TestRandomSearchFixed();
463             //links.Create64BillionLinksInParallel();
464             links.TestEachFunction();
465             //links.TestForeach();
466             //links.TestParallelForeach();
467         }
468
469         links.TestDeletionOfAllLinks();
470
471     }
472     catch (Exception ex)
473     {
474         ex.WriteToConsole();
475     }
476 }
477 */
478
479 /*

```

```

472     public static void TestLinksInSteps()
473     {
474         const long gibibyte = 1024 * 1024 * 1024;
475         const long mebibyte = 1024 * 1024;
476
477         var totalLinksToCreate = gibibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
478         var linksStep = 102 * mebibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
479
480         var creationMeasurements = new List<TimeSpan>();
481         var searchMeasurements = new List<TimeSpan>();
482         var deletionMeasurements = new List<TimeSpan>();
483
484         GetBaseRandomLoopOverhead(linksStep);
485         GetBaseRandomLoopOverhead(linksStep);
486
487         var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
488
489         ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
490
491         var loops = totalLinksToCreate / linksStep;
492
493         for (int i = 0; i < loops; i++)
494         {
495             creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
496             searchMeasurements.Add(Measure(() => links.RunRandomSearches(linksStep)));
497
498             Console.WriteLine("\rC + S {0}/{1}", i + 1, loops);
499         }
500
501         ConsoleHelpers.Debug();
502
503         for (int i = 0; i < loops; i++)
504         {
505             deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
506
507             Console.WriteLine("\rD {0}/{1}", i + 1, loops);
508         }
509
510         ConsoleHelpers.Debug();
511
512         ConsoleHelpers.Debug("C S D");
513
514         for (int i = 0; i < loops; i++)
515         {
516             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
↪ searchMeasurements[i], deletionMeasurements[i]);
517         }
518
519         ConsoleHelpers.Debug("C S D (no overhead)");
520
521         for (int i = 0; i < loops; i++)
522         {
523             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
↪ searchMeasurements[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
524         }
525
526         ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
↪ links.Total);
527     }
528
529     private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
↪ amountToCreate)
530     {
531         for (long i = 0; i < amountToCreate; i++)
532             links.Create(0, 0);
533     }
534
535     private static TimeSpan GetBaseRandomLoopOverhead(long loops)
536     {
537         return Measure(() =>
538         {
539             ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
540             ulong result = 0;
541             for (long i = 0; i < loops; i++)
542             {
543                 var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
544                 var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);

```

```

545         result += max_value + source + target;
546     }
547     Global.Trash = result;
548 });
549 }
550 */
551
552 [Fact(Skip = "performance test")]
553 public static void GetSourceTest()
554 {
555     using (var scope = new TempLinksTestScope())
556     {
557         var links = scope.Links;
558         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
559             ↪ Iterations);
560
561         ulong counter = 0;
562
563         //var firstLink = links.First();
564         // Создаём одну связь, из которой будет производить считывание
565         var firstLink = links.Create();
566
567         var sw = Stopwatch.StartNew();
568
569         // Тестируем саму функцию
570         for (ulong i = 0; i < Iterations; i++)
571         {
572             counter += links.GetSource(firstLink);
573         }
574
575         var elapsedTime = sw.Elapsed;
576
577         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
578
579         // Удаляем связь, из которой производилось считывание
580         links.Delete(firstLink);
581
582         ConsoleHelpers.Debug(
583             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
584             ↪ second), counter result: {3}",
585             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
586     }
587
588 [Fact(Skip = "performance test")]
589 public static void GetSourceInParallel()
590 {
591     using (var scope = new TempLinksTestScope())
592     {
593         var links = scope.Links;
594         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
595             ↪ parallel.", Iterations);
596
597         long counter = 0;
598
599         //var firstLink = links.First();
600         var firstLink = links.Create();
601
602         var sw = Stopwatch.StartNew();
603
604         // Тестируем саму функцию
605         Parallel.For(0, Iterations, x =>
606         {
607             Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
608             //Interlocked.Increment(ref counter);
609         });
610
611         var elapsedTime = sw.Elapsed;
612
613         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
614
615         links.Delete(firstLink);
616
617         ConsoleHelpers.Debug(
618             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
619             ↪ second), counter result: {3}",
620             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
621     }

```

```

620     }
621
622     [Fact(Skip = "performance test")]
623     public static void TestGetTarget()
624     {
625         using (var scope = new TempLinksTestScope())
626         {
627             var links = scope.Links;
628             ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
629                 ↪ Iterations);
630
631             ulong counter = 0;
632
633             //var firstLink = links.First();
634             var firstLink = links.Create();
635
636             var sw = Stopwatch.StartNew();
637
638             for (ulong i = 0; i < Iterations; i++)
639             {
640                 counter += links.GetTarget(firstLink);
641             }
642
643             var elapsedTime = sw.Elapsed;
644
645             var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
646
647             links.Delete(firstLink);
648
649             ConsoleHelpers.Debug(
650                 "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
651                 ↪ second), counter result: {3}",
652                 Iterations, elapsedTime, (long)iterationsPerSecond, counter);
653         }
654     }
655
656     [Fact(Skip = "performance test")]
657     public static void TestGetTargetInParallel()
658     {
659         using (var scope = new TempLinksTestScope())
660         {
661             var links = scope.Links;
662             ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
663                 ↪ parallel.", Iterations);
664
665             long counter = 0;
666
667             //var firstLink = links.First();
668             var firstLink = links.Create();
669
670             var sw = Stopwatch.StartNew();
671
672             Parallel.For(0, Iterations, x =>
673             {
674                 Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
675                 //Interlocked.Increment(ref counter);
676             });
677
678             var elapsedTime = sw.Elapsed;
679
680             var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
681
682             links.Delete(firstLink);
683
684             ConsoleHelpers.Debug(
685                 "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
686                 ↪ second), counter result: {3}",
687                 Iterations, elapsedTime, (long)iterationsPerSecond, counter);
688         }
689     }
690
691     // TODO: Заполнить базу данных перед тестом
692     /*
693     [Fact]
694     public void TestRandomSearchFixed()
695     {
696         var tempFilename = Path.GetTempFileName();
697
698         using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
699             ↪ 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
735             ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
↪ links.Count());
736
737             var sw = Stopwatch.StartNew();
738
739             for (var i = iterations; i > 0; i--)
740             {
741                 var linksAddressRange = new
↪ Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);
742
743                 var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
744                 var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
745
746                 counter += links.SearchOrDefault(source, target);
747             }
748
749             var elapsedTime = sw.Elapsed;
750
751             var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
752
753             ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪ Iterations per second), c: {3}",
↪ iterations, elapsedTime, (long)iterationsPerSecond, counter);
754         }
755     }
756
757     [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
758     public static void TestEach()
759     {
760         using (var scope = new TempLinksTestScope())
761         {
762             var links = scope.Links;
763
764             var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
765
766

```

```

767         ConsoleHelpers.Debug("Testing Each function.");
768
769         var sw = Stopwatch.StartNew();
770
771         links.Each(counter.IncrementAndReturnTrue);
772
773         var elapsedTime = sw.Elapsed;
774
775         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
776
777         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
↪         links per second)",
            counter, elapsedTime, (long)linksPerSecond);
778     }
779 }
780
781 /*
782 [Fact]
783 public static void TestForeach()
784 {
785     var tempFilename = Path.GetTempFileName();
786
787     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪     DefaultLinksSizeStep))
788     {
789         ulong counter = 0;
790
791         ConsoleHelpers.Debug("Testing foreach through links.");
792
793         var sw = Stopwatch.StartNew();
794
795         //foreach (var link in links)
796         //{
797             //    counter++;
798         //}
799
800         var elapsedTime = sw.Elapsed;
801
802         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
803
804         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
↪     links per second)", counter, elapsedTime, (long)linksPerSecond);
805     }
806
807     File.Delete(tempFilename);
808 }
809 */
810
811 /*
812 [Fact]
813 public static void TestParallelForeach()
814 {
815     var tempFilename = Path.GetTempFileName();
816
817     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪     DefaultLinksSizeStep))
818     {
819
820         long counter = 0;
821
822         ConsoleHelpers.Debug("Testing parallel foreach through links.");
823
824         var sw = Stopwatch.StartNew();
825
826         //Parallel.ForEach((IEnumerable<ulong>)links, x =>
827         //{
828             //    Interlocked.Increment(ref counter);
829         //});
830
831         var elapsedTime = sw.Elapsed;
832
833         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
834
835         ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↪     {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
836     }
837
838     File.Delete(tempFilename);
839 }
840 */
841

```



```

842 [Fact(Skip = "performance test")]
843 public static void Create64BillionLinks()
844 {
845     using (var scope = new TempLinksTestScope())
846     {
847         var links = scope.Links;
848         var linksBeforeTest = links.Count();
849
850         long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;
851
852         ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
853
854         var elapsedTime = Performance.Measure(() =>
855         {
856             for (long i = 0; i < linksToCreate; i++)
857             {
858                 links.Create();
859             }
860         });
861
862         var linksCreated = links.Count() - linksBeforeTest;
863         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
864
865         ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
866
867         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
868             ↪ 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
900 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
901 public static void TestDeletionOfAllLinks()
902 {
903     using (var scope = new TempLinksTestScope())
904     {
905         var links = scope.Links;
906         var linksBeforeTest = links.Count();
907
908         ConsoleHelpers.Debug("Deleting all links");
909
910         var elapsedTime = Performance.Measure(links.DeleteAll);
911
912         var linksDeleted = linksBeforeTest - links.Count();
913         var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
914
915         ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
916             ↪ linksDeleted, elapsedTime,
917             (long)linksPerSecond);
918     }
919 }

```

```
#endregion
```

```
919 }  
920 }  
921 }
```

1.126 ./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]));  
39                    Assert.Equal(numbers[i],  
40                    ↪ fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));  
41                }  
42            }  
43        }  
44    }  
45 }
```

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

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

```

28     var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
29     var powerOf2ToUnaryNumberConverter = new
    ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
30     var addressToUnaryNumberConverter = new
    ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
31     var unaryNumberToAddressConverter = new
    ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
    ↪ powerOf2ToUnaryNumberConverter);
32     TestCharAndUnicodeSymbolConverters(links, meaningRoot,
    ↪ addressToUnaryNumberConverter, unaryNumberToAddressConverter);
33 }
34 }
35
36 [Fact]
37 public static void CharAndRawNumberUnicodeSymbolConvertersTest()
38 {
39     using (var scope = new Scope<Types<HeapResizableDirectMemory,
    ↪ UnitedMemoryLinks<ulong>>>())
40     {
41         var links = scope.Use<ILinks<ulong>>();
42         var meaningRoot = links.CreatePoint();
43         var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
44         var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
45         TestCharAndUnicodeSymbolConverters(links, meaningRoot,
    ↪ addressToRawNumberConverter, rawNumberToAddressConverter);
46     }
47 }
48
49 private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
    ↪ meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
    ↪ numberToAddressConverter)
50 {
51     var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
52     var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
    ↪ addressToNumberConverter, unicodeSymbolMarker);
53     var originalCharacter = 'H';
54     var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
55     var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
    ↪ unicodeSymbolMarker);
56     var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
    ↪ numberToAddressConverter, unicodeSymbolCriterionMatcher);
57     var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
58     Assert.Equal(originalCharacter, resultingCharacter);
59 }
60
61 [Fact]
62 public static void StringAndUnicodeSequenceConvertersTest()
63 {
64     using (var scope = new TempLinksTestScope())
65     {
66         var links = scope.Links;
67
68         var itself = links.Constants.Itself;
69
70         var meaningRoot = links.CreatePoint();
71         var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
72         var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
73         var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
74         var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
75         var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
76
77         var powerOf2ToUnaryNumberConverter = new
    ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
78         var addressToUnaryNumberConverter = new
    ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
79         var charToUnicodeSymbolConverter = new
    ↪ CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
    ↪ unicodeSymbolMarker);
80
81         var unaryNumberToAddressConverter = new
    ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
    ↪ powerOf2ToUnaryNumberConverter);
82         var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
83         var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
    ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
84         var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
    ↪ frequencyPropertyMarker, frequencyMarker);

```

```

85     var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
86         ↪ frequencyPropertyOperator, frequencyIncrementer);
87     var linkToItsFrequencyNumberConverter = new
88         ↪ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
89         ↪ unaryNumberToAddressConverter);
90     var sequenceToItsLocalElementLevelsConverter = new
91         ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
92         ↪ linkToItsFrequencyNumberConverter);
93     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
94         ↪ sequenceToItsLocalElementLevelsConverter);
95
96     var stringToUnicodeSequenceConverter = new
97         ↪ StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
98         ↪ index, optimalVariantConverter, unicodeSequenceMarker);
99
100     var originalString = "Hello";
101
102     var unicodeSequenceLink =
103         ↪ stringToUnicodeSequenceConverter.Convert(originalString);
104
105     var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
106         ↪ unicodeSymbolMarker);
107     var unicodeSymbolToCharConverter = new
108         ↪ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
109         ↪ unicodeSymbolCriterionMatcher);
110
111     var unicodeSequenceCriterionMatcher = new TargetMatcher<ulong>(links,
112         ↪ unicodeSequenceMarker);
113
114     var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
115         ↪ unicodeSymbolCriterionMatcher.IsMatched);
116
117     var unicodeSequenceToStringConverter = new
118         ↪ UnicodeSequenceToStringConverter<ulong>(links,
119         ↪ unicodeSequenceCriterionMatcher, sequenceWalker,
120         ↪ unicodeSymbolToCharConverter);
121
122     var resultingString =
123         ↪ unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
124
125     Assert.Equal(originalString, resultingString);
126 }
127 }
128 }

```

Index

`./csharp/Platform.Data.Doublets.Tests/GenericLinksTests.cs`, 171
`./csharp/Platform.Data.Doublets.Tests/ILinksExtensionsTests.cs`, 172
`./csharp/Platform.Data.Doublets.Tests/LinksConstantsTests.cs`, 172
`./csharp/Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs`, 173
`./csharp/Platform.Data.Doublets.Tests/ReadSequenceTests.cs`, 176
`./csharp/Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs`, 177
`./csharp/Platform.Data.Doublets.Tests/ScopeTests.cs`, 177
`./csharp/Platform.Data.Doublets.Tests/SequencesTests.cs`, 178
`./csharp/Platform.Data.Doublets.Tests/SplitMemoryGenericLinksTests.cs`, 193
`./csharp/Platform.Data.Doublets.Tests/TempLinksTestScope.cs`, 194
`./csharp/Platform.Data.Doublets.Tests/TestExtensions.cs`, 194
`./csharp/Platform.Data.Doublets.Tests/UInt64LinksTests.cs`, 197
`./csharp/Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs`, 210
`./csharp/Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs`, 210
`./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`, 29
`./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`, 30
`./csharp/Platform.Data.Doublets/Memory/LinksHeader.cs`, 31
`./csharp/Platform.Data.Doublets/Memory/Split/Generic/ExternalLinksSizeBalancedTreeMethodsBase.cs`, 31
`./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`, 36
`./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`, 52
`./csharp/Platform.Data.Doublets/Memory/Split/RawLinkIndexPart.cs`, 53
`./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksAvlBalancedTreeMethodsBase.cs`, 54
`./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSizeBalancedTreeMethodsBase.cs`, 58
`./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesAvlBalancedTreeMethods.cs`, 61
`./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksSourcesSizeBalancedTreeMethods.cs`, 62
`./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsAvlBalancedTreeMethods.cs`, 63
`./csharp/Platform.Data.Doublets/Memory/United/Generic/LinksTargetsSizeBalancedTreeMethods.cs`, 64
`./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinks.cs`, 65
`./csharp/Platform.Data.Doublets/Memory/United/Generic/UnitedMemoryLinksBase.cs`, 66
`./csharp/Platform.Data.Doublets/Memory/United/Generic/UnusedLinksListMethods.cs`, 74
`./csharp/Platform.Data.Doublets/Memory/United/RawLink.cs`, 75
`./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs`, 75
`./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSizeBalancedTreeMethodsBase.cs`, 77

./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesAvlBalancedTreeMethods.cs, 78
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksSourcesSizeBalancedTreeMethods.cs, 79
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsAvlBalancedTreeMethods.cs, 80
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs, 81
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnitedMemoryLinks.cs, 82
./csharp/Platform.Data.Doublets/Memory/United/Specific/UInt64UnusedLinksListMethods.cs, 84
./csharp/Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs, 84
./csharp/Platform.Data.Doublets/Numbers/Unary/LinkToltsFrequencyNumberConveter.cs, 85
./csharp/Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs, 86
./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs, 86
./csharp/Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs, 87
./csharp/Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 88
./csharp/Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 89
./csharp/Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 90
./csharp/Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 91
./csharp/Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 94
./csharp/Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 94
./csharp/Platform.Data.Doublets/Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 96
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs, 96
./csharp/Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs, 97
./csharp/Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 97
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 98
./csharp/Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 98
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 101
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 102
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToltsFrequencyValueConverter.cs, 103
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 103
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 104
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 104
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 105
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 105
./csharp/Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 105
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 106
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 107
./csharp/Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 107
./csharp/Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 108
./csharp/Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 109
./csharp/Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 109
./csharp/Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 110
./csharp/Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 110
./csharp/Platform.Data.Doublets/Sequences/Indexes/Unindex.cs, 111
./csharp/Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 111
./csharp/Platform.Data.Doublets/Sequences/Sequences.cs, 138
./csharp/Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 149
./csharp/Platform.Data.Doublets/Sequences/SequencesOptions.cs, 150
./csharp/Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 152
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 152
./csharp/Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 153
./csharp/Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 155
./csharp/Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 155
./csharp/Platform.Data.Doublets/Stacks/Stack.cs, 156
./csharp/Platform.Data.Doublets/Stacks/StackExtensions.cs, 157
./csharp/Platform.Data.Doublets/SynchronizedLinks.cs, 157
./csharp/Platform.Data.Doublets/UInt64LinksExtensions.cs, 158
./csharp/Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 160
./csharp/Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 166
./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 166
./csharp/Platform.Data.Doublets/Unicode/StringToUnicodeSymbolsListConverter.cs, 167
./csharp/Platform.Data.Doublets/Unicode/UnicodeMap.cs, 168
./csharp/Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 170
./csharp/Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 171