1.1 ./Platform.Data.Doublets/Decorators/LinksCascadeUniquenessAndUsagesResolver.cs

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

using System.Runtime.CompilerServices;

namespace Platform.Data.Doublets.Decorators
{
    public class LinksCascadeUniquenessAndUsagesResolver<TLink> : LinksUniquenessResolver<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksCascadeUniquenessAndUsagesResolver(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
        ↪  newLinkAddress)
        {
            // Use Facade (the last decorator) to ensure recursion working correctly
            Facade.MergeUsages(oldLinkAddress, newLinkAddress);
            return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
        }
    }
}
```

1.2 ./Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    /// <remarks>
    /// <para>Must be used in conjunction with NonNullContentsLinkDeletionResolver.</para>
    /// <para>Должен использоваться вместе с NonNullContentsLinkDeletionResolver.</para>
    /// </remarks>
    public class LinksCascadeUsagesResolver<TLink> : LinksDecoratorBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksCascadeUsagesResolver(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override void Delete(IList<TLink> restrictions)
        {
            var linkIndex = restrictions[Constants.IndexPart];
            // Use Facade (the last decorator) to ensure recursion working correctly
            Facade.DeleteAllUsages(linkIndex);
            Links.Delete(linkIndex);
        }
    }
}
```

1.3 ./Platform.Data.Doublets/Decorators/LinksDecoratorBase.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    public abstract class LinksDecoratorBase<TLink> : LinksOperatorBase<TLink>, ILinks<TLink>
    {
        private ILinks<TLink> _facade;

        public LinksConstants<TLink> Constants { get; }

        public ILinks<TLink> Facade
        {
            get => _facade;
            set
            {
                _facade = value;
                if (Links is LinksDecoratorBase<TLink> decorator)
                {
                    decorator.Facade = value;
                }
                else if (Links is LinksDisposableDecoratorBase<TLink> disposableDecorator)
                {
```

```
                    disposableDecorator.Facade = value;
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksDecoratorBase(ILinks<TLink> links) : base(links)
        {
            Constants = links.Constants;
            Facade = this;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Count(IList<TLink> restrictions) => Links.Count(restrictions);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        ↪  => Links.Each(handler, restrictions);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Create(IList<TLink> restrictions) => Links.Create(restrictions);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
        ↪  Links.Update(restrictions, substitution);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual void Delete(IList<TLink> restrictions) => Links.Delete(restrictions);
    }
}
```

1.4 ./Platform.Data.Doublets/Decorators/LinksDisposableDecoratorBase.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Disposables;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    public abstract class LinksDisposableDecoratorBase<TLink> : DisposableBase, ILinks<TLink>
    {
        private ILinks<TLink> _facade;

        public LinksConstants<TLink> Constants { get; }

        public ILinks<TLink> Links { get; }

        public ILinks<TLink> Facade
        {
            get => _facade;
            set
            {
                _facade = value;
                if (Links is LinksDecoratorBase<TLink> decorator)
                {
                    decorator.Facade = value;
                }
                else if (Links is LinksDisposableDecoratorBase<TLink> disposableDecorator)
                {
                    disposableDecorator.Facade = value;
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksDisposableDecoratorBase(ILinks<TLink> links)
        {
            Links = links;
            Constants = links.Constants;
            Facade = this;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Count(IList<TLink> restrictions) => Links.Count(restrictions);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        ↪  => Links.Each(handler, restrictions);
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Create(IList<TLink> restrictions) => Links.Create(restrictions);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
        ↪  Links.Update(restrictions, substitution);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual void Delete(IList<TLink> restrictions) => Links.Delete(restrictions);

        protected override bool AllowMultipleDisposeCalls => true;

        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                Links.DisposeIfPossible();
            }
        }
    }
}
```

1.5 ./Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    // TODO: Make LinksExternalReferenceValidator. A layer that checks each link to exist or to
    ↪  be external (hybrid link's raw number).
    public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        {
            Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
            return Links.Each(handler, restrictions);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
            Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
            Links.EnsureInnerReferenceExists(substitution, nameof(substitution));
            return Links.Update(restrictions, substitution);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override void Delete(IList<TLink> restrictions)
        {
            var link = restrictions[Constants.IndexPart];
            Links.EnsureLinkExists(link, nameof(link));
            Links.Delete(link);
        }
    }
}
```

1.6 ./Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    public class LinksItselfConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksItselfConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        {
            var constants = Constants;
            var itselfConstant = constants.Itself;
            var indexPartConstant = constants.IndexPart;
            var sourcePartConstant = constants.SourcePart;
            var targetPartConstant = constants.TargetPart;
            var restrictionsCount = restrictions.Count;
            if (!_equalityComparer.Equals(constants.Any, itselfConstant)
             && (((restrictionsCount > indexPartConstant) &&
              ↪  _equalityComparer.Equals(restrictions[indexPartConstant], itselfConstant))
             || ((restrictionsCount > sourcePartConstant) &&
              ↪  _equalityComparer.Equals(restrictions[sourcePartConstant], itselfConstant))
             || ((restrictionsCount > targetPartConstant) &&
              ↪  _equalityComparer.Equals(restrictions[targetPartConstant], itselfConstant))))
            {
                // Itself constant is not supported for Each method right now, skipping execution
                return constants.Continue;
            }
            return Links.Each(handler, restrictions);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
         ↪  Links.Update(restrictions, Links.ResolveConstantAsSelfReference(Constants.Itself,
         ↪  restrictions, substitution));
    }
}
```

1.7 ./Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    /// <remarks>
    /// Not practical if newSource and newTarget are too big.
    /// To be able to use practical version we should allow to create link at any specific
     ↪  location inside ResizableDirectMemoryLinks.
    /// This in turn will require to implement not a list of empty links, but a list of ranges
     ↪  to store it more efficiently.
    /// </remarks>
    public class LinksNonExistentDependenciesCreator<TLink> : LinksDecoratorBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksNonExistentDependenciesCreator(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            var constants = Constants;
            Links.EnsureCreated(substitution[constants.SourcePart],
             ↪  substitution[constants.TargetPart]);
            return Links.Update(restrictions, substitution);
        }
    }
}
```

1.8 ./Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    public class LinksNullConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksNullConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Create(IList<TLink> restrictions)
```

```
15          {
16              var link = Links.Create();
17              return Links.Update(link, link, link);
18          }
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
            ↪  Links.Update(restrictions, Links.ResolveConstantAsSelfReference(Constants.Null,
            ↪  restrictions, substitution));
22      }
23  }
```

1.9 ./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 =
            ↪  EqualityComparer<TLink>.Default;
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          public LinksUniquenessResolver(ILinks<TLink> links) : base(links) { }
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
17          {
18              var newLinkAddress = Links.SearchOrDefault(substitution[Constants.SourcePart],
                ↪  substitution[Constants.TargetPart]);
19              if (_equalityComparer.Equals(newLinkAddress, default))
20              {
21                  return Links.Update(restrictions, substitution);
22              }
23              return ResolveAddressChangeConflict(restrictions[Constants.IndexPart],
                ↪  newLinkAddress);
24          }
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
            ↪  newLinkAddress)
28          {
29              if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
                ↪  Links.Exists(oldLinkAddress))
30              {
31                  Facade.Delete(oldLinkAddress);
32              }
33              return newLinkAddress;
34          }
35      }
36  }
```

1.10 ./Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs

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

1.11 ./Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    public class LinksUsagesValidator<TLink> : LinksDecoratorBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksUsagesValidator(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
        {
            Links.EnsureNoUsages(restrictions[Constants.IndexPart]);
            return Links.Update(restrictions, substitution);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override void Delete(IList<TLink> restrictions)
        {
            var link = restrictions[Constants.IndexPart];
            Links.EnsureNoUsages(link);
            Links.Delete(link);
        }
    }
}
```

1.12 ./Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    public class NonNullContentsLinkDeletionResolver<TLink> : LinksDecoratorBase<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public NonNullContentsLinkDeletionResolver(ILinks<TLink> links) : base(links) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override void Delete(IList<TLink> restrictions)
        {
            var linkIndex = restrictions[Constants.IndexPart];
            Links.EnforceResetValues(linkIndex);
            Links.Delete(linkIndex);
        }
    }
}
```

1.13 ./Platform.Data.Doublets/Decorators/UInt64Links.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Decorators
{
    /// <summary>
    /// Представляет объект для работы с базой данных (файлом) в формате Links (массива связей).
    /// </summary>
    /// <remarks>
    /// Возможные оптимизации:
    /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
    ///     + меньше объём БД
    ///     - меньше производительность
    ///     - больше ограничение на количество связей в БД)
    /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
    ///     + меньше объём БД
    ///     - больше сложность
    ///
    /// 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
    /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
    ↪  (битовыми строками) - вариант матрицы (выстраеваемой лениво).
```

```
23      ///
24      /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
        ↪  выбрасываться только при #if DEBUG
25      /// </remarks>
26      public class UInt64Links : LinksDisposableDecoratorBase<ulong>
27      {
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          public UInt64Links(ILinks<ulong> links) : base(links) { }
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          public override ulong Create(IList<ulong> restrictions) => Links.CreatePoint();
33
34          public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
35          {
36              var constants = Constants;
37              var indexPartConstant = constants.IndexPart;
38              var updatedLink = restrictions[indexPartConstant];
39              var sourcePartConstant = constants.SourcePart;
40              var newSource = substitution[sourcePartConstant];
41              var targetPartConstant = constants.TargetPart;
42              var newTarget = substitution[targetPartConstant];
43              var nullConstant = constants.Null;
44              var existedLink = nullConstant;
45              var itselfConstant = constants.Itself;
46              if (newSource != itselfConstant && newTarget != itselfConstant)
47              {
48                  existedLink = Links.SearchOrDefault(newSource, newTarget);
49              }
50              if (existedLink == nullConstant)
51              {
52                  var before = Links.GetLink(updatedLink);
53                  if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
                    ↪  newTarget)
54                  {
55                      Links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
                        ↪  newSource,
56                                                newTarget == itselfConstant ? updatedLink :
                                                  ↪  newTarget);
57                  }
58                  return updatedLink;
59              }
60              else
61              {
62                  return Facade.MergeAndDelete(updatedLink, existedLink);
63              }
64          }
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          public override void Delete(IList<ulong> restrictions)
68          {
69              var linkIndex = restrictions[Constants.IndexPart];
70              Links.EnforceResetValues(linkIndex);
71              Facade.DeleteAllUsages(linkIndex);
72              Links.Delete(linkIndex);
73          }
74      }
75  }
```

1.14 ./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
        ↪  by itself. But can cause creation (update from nothing) or deletion (update to nothing).
15      ///
16      /// TODO: Decide to change to IDoubletLinks or not to change. (Better to create
        ↪  DefaultUniLinksBase, that contains logic itself and can be implemented using both
        ↪  IDoubletLinks and ILinks.)
17      /// </remarks>
```

```
18      internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
19      {
20          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
21
22          public UniLinks(ILinks<TLink> links) : base(links) { }
23
24          private struct Transition
25          {
26              public IList<TLink> Before;
27              public IList<TLink> After;
28
29              public Transition(IList<TLink> before, IList<TLink> after)
30              {
31                  Before = before;
32                  After = after;
33              }
34          }
35
36          //public static readonly TLink NullConstant = Use<LinksConstants<TLink>>.Single.Null;
37          //public static readonly IReadOnlyList<TLink> NullLink = new
            ↪ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
            ↪ });
38
39          // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
            ↪ (Links-Expression)
40          public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
            ↪ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
            ↪ substitutedHandler)
41          {
42              ////List<Transition> transitions = null;
43              ////if (!restriction.IsNullOrEmpty())
44              ////{
45              ////    // Есть причина делать проход (чтение)
46              ////    if (matchedHandler != null)
47              ////    {
48              ////        if (!substitution.IsNullOrEmpty())
49              ////        {
50              ////            // restriction => { 0, 0, 0 } | { 0 } // Create
51              ////            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
                ↪ Create / Update
52              ////            // substitution => { 0, 0, 0 } | { 0 } // Delete
53              ////            transitions = new List<Transition>();
54              ////            if (Equals(substitution[Constants.IndexPart], Constants.Null))
55              ////            {
56              ////                // If index is Null, that means we always ignore every other
                ↪ value (they are also Null by definition)
57              ////                var matchDecision = matchedHandler(, NullLink);
58              ////                if (Equals(matchDecision, Constants.Break))
59              ////                    return false;
60              ////                if (!Equals(matchDecision, Constants.Skip))
61              ////                    transitions.Add(new Transition(matchedLink, newValue));
62              ////            }
63              ////            else
64              ////            {
65              ////                Func<T, bool> handler;
66              ////                handler = link =>
67              ////                {
68              ////                    var matchedLink = Memory.GetLinkValue(link);
69              ////                    var newValue = Memory.GetLinkValue(link);
70              ////                    newValue[Constants.IndexPart] = Constants.Itself;
71              ////                    newValue[Constants.SourcePart] =
                ↪ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
                ↪ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
72              ////                    newValue[Constants.TargetPart] =
                ↪ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
                ↪ 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              ////        }
```

```
            ////        else
            ////        {
            ////            Func<T, bool> handler = link =>
            ////            {
            ////                var matchedLink = Memory.GetLinkValue(link);
            ////                var matchDecision = matchedHandler(matchedLink, matchedLink);
            ////                return !Equals(matchDecision, Constants.Break);
            ////            };
            ////            if (!Memory.Each(handler, restriction))
            ////                return Constants.Break;
            ////        }
            ////    }
            ////    else
            ////    {
            ////        if (substitution != null)
            ////        {
            ////            transitions = new List<IList<T>>();
            ////            Func<T, bool> handler = link =>
            ////            {
            ////                var matchedLink = Memory.GetLinkValue(link);
            ////                transitions.Add(matchedLink);
            ////                return true;
            ////            };
            ////            if (!Memory.Each(handler, restriction))
            ////                return Constants.Break;
            ////        }
            ////        else
            ////        {
            ////            return Constants.Continue;
            ////        }
            ////    }
            ////}
            ////if (substitution != null)
            ////{
            ////    // Есть причина делать замену (запись)
            ////    if (substitutedHandler != null)
            ////    {
            ////    }
            ////    else
            ////    {
            ////    }
            ////}
            ////return Constants.Continue;

            //if (restriction.IsNullOrEmpty()) // Create
            //{
            //    substitution[Constants.IndexPart] = Memory.AllocateLink();
            //    Memory.SetLinkValue(substitution);
            //}
            //else if (substitution.IsNullOrEmpty()) // Delete
            //{
            //    Memory.FreeLink(restriction[Constants.IndexPart]);
            //}
            //else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
            //{
            //    // No need to collect links to list
            //    // Skip == Continue
            //    // No need to check substituedHandler
            //    if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
            ↪  Constants.Break), restriction))
            //        return Constants.Break;
            //}
            //else // Update
            //{
            //    //List<IList<T>> matchedLinks = null;
            //    if (matchedHandler != null)
            //    {
            //        matchedLinks = new List<IList<T>>();
            //        Func<T, bool> handler = link =>
            //        {
            //            var matchedLink = Memory.GetLinkValue(link);
            //            var matchDecision = matchedHandler(matchedLink);
            //            if (Equals(matchDecision, Constants.Break))
            //                return false;
            //            if (!Equals(matchDecision, Constants.Skip))
            //                matchedLinks.Add(matchedLink);
            //            return true;
            //        };
```

```
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
                ↪  substitution handling.
174             //                var substitutedDecision = substitutedHandler(matchedLink,
                ↪  newValue);
175             //                if (Equals(substitutedDecision, Constants.Break))
176             //                    return Constants.Break;
177             //                if (Equals(substitutedDecision, Constants.Continue))
178             //                {
179             //                    // Actual update here
180             //                    Memory.SetLinkValue(newValue);
181             //                }
182             //                if (Equals(substitutedDecision, Constants.Skip))
183             //                {
184             //                    // Cancel the update. TODO: decide use separate Cancel
                ↪  constant or Skip is enough?
185             //                }
186             //            }
187             //        }
188             //    }
189             //}
190             return Constants.Continue;
191         }
192 
193         public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
            ↪  matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
            ↪  substitutionHandler)
194         {
195             if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
196             {
197                 return Constants.Continue;
198             }
199             else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
                ↪  Check if it is a correct condition
200             {
201                 // Or it only applies to trigger without matchHandler.
202                 throw new NotImplementedException();
203             }
204             else if (!substitution.IsNullOrEmpty()) // Creation
205             {
206                 var before = Array.Empty<TLink>();
207                 // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
                    ↪  (пройти мимо) или пустить (взять)?
208                 if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                    ↪  Constants.Break))
209                 {
210                     return Constants.Break;
211                 }
212                 var after = (IList<TLink>)substitution.ToArray();
213                 if (_equalityComparer.Equals(after[0], default))
214                 {
215                     var newLink = Links.Create();
216                     after[0] = newLink;
217                 }
218                 if (substitution.Count == 1)
219                 {
220                     after = Links.GetLink(substitution[0]);
221                 }
222                 else if (substitution.Count == 3)
223                 {
224                     //Links.Create(after);
225                 }
226                 else
227                 {
228                     throw new NotSupportedException();
229                 }
230                 if (matchHandler != null)
```

```
231                 {
232                     return substitutionHandler(before, after);
233                 }
234                 return Constants.Continue;
235             }
236             else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
237             {
238                 if (patternOrCondition.Count == 1)
239                 {
240                     var linkToDelete = patternOrCondition[0];
241                     var before = Links.GetLink(linkToDelete);
242                     if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                        ↪ Constants.Break))
243                     {
244                         return Constants.Break;
245                     }
246                     var after = Array.Empty<TLink>();
247                     Links.Update(linkToDelete, Constants.Null, Constants.Null);
248                     Links.Delete(linkToDelete);
249                     if (matchHandler != null)
250                     {
251                         return substitutionHandler(before, after);
252                     }
253                     return Constants.Continue;
254                 }
255                 else
256                 {
257                     throw new NotSupportedException();
258                 }
259             }
260             else // Replace / Update
261             {
262                 if (patternOrCondition.Count == 1) //-V3125
263                 {
264                     var linkToUpdate = patternOrCondition[0];
265                     var before = Links.GetLink(linkToUpdate);
266                     if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
                        ↪ Constants.Break))
267                     {
268                         return Constants.Break;
269                     }
270                     var after = (IList<TLink>)substitution.ToArray(); //-V3125
271                     if (_equalityComparer.Equals(after[0], default))
272                     {
273                         after[0] = linkToUpdate;
274                     }
275                     if (substitution.Count == 1)
276                     {
277                         if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
278                         {
279                             after = Links.GetLink(substitution[0]);
280                             Links.Update(linkToUpdate, Constants.Null, Constants.Null);
281                             Links.Delete(linkToUpdate);
282                         }
283                     }
284                     else if (substitution.Count == 3)
285                     {
286                         //Links.Update(after);
287                     }
288                     else
289                     {
290                         throw new NotSupportedException();
291                     }
292                     if (matchHandler != null)
293                     {
294                         return substitutionHandler(before, after);
295                     }
296                     return Constants.Continue;
297                 }
298                 else
299                 {
300                     throw new NotSupportedException();
301                 }
302             }
303         }
304 
305         /// <remarks>
306         /// IList[IList[IList[T]]]
```

```
307             /// |      |      |      |||
308             /// |      |      |      ------ ||
309             /// |      |      |       link  ||
310             /// |      |      -------------- |
311             /// |      |         change      |
312             ///  ---------------------
313             ///         changes
314             /// </remarks>
315             public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink>
                ↪  substitution)
316             {
317                 var changes = new List<IList<IList<TLink>>>();
318                 Trigger(condition, AlwaysContinue, substitution, (before, after) =>
319                 {
320                     var change = new[] { before, after };
321                     changes.Add(change);
322                     return Constants.Continue;
323                 });
324                 return changes;
325             }
326
327             private TLink AlwaysContinue(IList<TLink> linkToMatch) => Constants.Continue;
328         }
329     }
```

1.15 ./Platform.Data.Doublets/Doublet.cs

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

1.16 ./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  }
```

1.17 ./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  }
```

1.18 ./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          public static void RunRandomCreations<TLink>(this ILinks<TLink> links, ulong
            ↪  amountOfCreations)
22          {
23              var random = RandomHelpers.Default;
24              var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
25              var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
26              for (var i = 0UL; i < amountOfCreations; i++)
27              {
28                  var linksAddressRange = new Range<ulong>(0,
                    ↪  addressToUInt64Converter.Convert(links.Count()));
29                  var source =
                    ↪  uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
30                  var target =
                    ↪  uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
31                  links.GetOrCreate(source, target);
32              }
33          }
34
35          public static void RunRandomSearches<TLink>(this ILinks<TLink> links, ulong
            ↪  amountOfSearches)
36          {
37              var random = RandomHelpers.Default;
38              var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
39              var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
40              for (var i = 0UL; i < amountOfSearches; i++)
41              {
42                  var linksAddressRange = new Range<ulong>(0,
                    ↪  addressToUInt64Converter.Convert(links.Count()));
43                  var source =
                    ↪  uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
44                  var target =
                    ↪  uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
45                  links.SearchOrDefault(source, target);
46              }
47          }
48
49          public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, ulong
            ↪  amountOfDeletions)
50          {
```

```
51              var random = RandomHelpers.Default;
52              var addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
53              var uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
54              var linksCount = addressToUInt64Converter.Convert(links.Count());
55              var min = amountOfDeletions > linksCount ? OUL : linksCount - amountOfDeletions;
56              for (var i = OUL; i < amountOfDeletions; i++)
57              {
58                  linksCount = addressToUInt64Converter.Convert(links.Count());
59                  if (linksCount <= min)
60                  {
61                      break;
62                  }
63                  var linksAddressRange = new Range<ulong>(min, linksCount);
64                  var link =
                    ↪  uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
65                  links.Delete(link);
66              }
67          }
68
69          public static void Delete<TLink>(this ILinks<TLink> links, TLink linkToDelete) =>
            ↪  links.Delete(new LinkAddress<TLink>(linkToDelete));
70
71          /// <remarks>
72          /// TODO: Возможно есть очень простой способ это сделать.
73          /// (Например просто удалить файл, или изменить его размер таким образом,
74          /// чтобы удалился весь контент)
75          /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
76          /// </remarks>
77          public static void DeleteAll<TLink>(this ILinks<TLink> links)
78          {
79              var equalityComparer = EqualityComparer<TLink>.Default;
80              var comparer = Comparer<TLink>.Default;
81              for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
                ↪  Arithmetic.Decrement(i))
82              {
83                  links.Delete(i);
84                  if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
85                  {
86                      i = links.Count();
87                  }
88              }
89          }
90
91          public static TLink First<TLink>(this ILinks<TLink> links)
92          {
93              TLink firstLink = default;
94              var equalityComparer = EqualityComparer<TLink>.Default;
95              if (equalityComparer.Equals(links.Count(), default))
96              {
97                  throw new InvalidOperationException("В хранилище нет связей.");
98              }
99              links.Each(links.Constants.Any, links.Constants.Any, link =>
100             {
101                 firstLink = link[links.Constants.IndexPart];
102                 return links.Constants.Break;
103             });
104             if (equalityComparer.Equals(firstLink, default))
105             {
106                 throw new InvalidOperationException("В процессе поиска по хранилищу не было
                    ↪  найдено связей.");
107             }
108             return firstLink;
109         }
110
111         #region Paths
112
113         /// <remarks>
114         /// TODO: Как так? Как то что ниже может быть корректно?
115         /// Скорее всего практически не применимо
116         /// Предполагалось, что можно было конвертировать формируемый в проходе через
            ↪  SequenceWalker
117         /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
118         /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
119         /// </remarks>
120         public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
            ↪  path)
121         {
122             var current = path[0];
123             //EnsureLinkExists(current, "path");
```

```
124             if (!links.Exists(current))
125             {
126                 return false;
127             }
128             var equalityComparer = EqualityComparer<TLink>.Default;
129             var constants = links.Constants;
130             for (var i = 1; i < path.Length; i++)
131             {
132                 var next = path[i];
133                 var values = links.GetLink(current);
134                 var source = values[constants.SourcePart];
135                 var target = values[constants.TargetPart];
136                 if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
                    ↪ next))
137                 {
138                     //throw new InvalidOperationException(string.Format("Невозможно выбрать
                        ↪ путь, так как и Source и Target совпадают с элементом пути {0}.", next));
139                     return false;
140                 }
141                 if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
                    ↪ target))
142                 {
143                     //throw new InvalidOperationException(string.Format("Невозможно продолжить
                        ↪ путь через элемент пути {0}", next));
144                     return false;
145                 }
146                 current = next;
147             }
148             return true;
149         }
150 
151         /// <remarks>
152         /// Может потребовать дополнительного стека для PathElement's при использовании
            ↪ SequenceWalker.
153         /// </remarks>
154         public static TLink GetByKeys<TLink>(this ILinks<TLink> links, TLink root, params int[]
            ↪ path)
155         {
156             links.EnsureLinkExists(root, "root");
157             var currentLink = root;
158             for (var i = 0; i < path.Length; i++)
159             {
160                 currentLink = links.GetLink(currentLink)[path[i]];
161             }
162             return currentLink;
163         }
164 
165         public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
            ↪ links, TLink root, ulong size, ulong index)
166         {
167             var constants = links.Constants;
168             var source = constants.SourcePart;
169             var target = constants.TargetPart;
170             if (!Platform.Numbers.Math.IsPowerOfTwo(size))
171             {
172                 throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
                    ↪ than powers of two are not supported.");
173             }
174             var path = new BitArray(BitConverter.GetBytes(index));
175             var length = Bit.GetLowestPosition(size);
176             links.EnsureLinkExists(root, "root");
177             var currentLink = root;
178             for (var i = length - 1; i >= 0; i--)
179             {
180                 currentLink = links.GetLink(currentLink)[path[i] ? target : source];
181             }
182             return currentLink;
183         }
184 
185         #endregion
186 
187         /// <summary>
188         /// Возвращает индекс указанной связи.
189         /// </summary>
190         /// <param name="links">Хранилище связей.</param>
191         /// <param name="link">Связь представленная списком, состоящим из её адреса и
            ↪ содержимого.</param>
192         /// <returns>Индекс начальной связи для указанной связи.</returns>
```

```
193         [MethodImpl(MethodImplOptions.AggressiveInlining)]
194         public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪ link[links.Constants.IndexPart];
195
196         /// <summary>
197         /// Возвращает индекс начальной (Source) связи для указанной связи.
198         /// </summary>
199         /// <param name="links">Хранилище связей.</param>
200         /// <param name="link">Индекс связи.</param>
201         /// <returns>Индекс начальной связи для указанной связи.</returns>
202         [MethodImpl(MethodImplOptions.AggressiveInlining)]
203         public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪ links.GetLink(link)[links.Constants.SourcePart];
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">Связь представленная списком, состоящим из её адреса и
            ↪ содержимого.</param>
210         /// <returns>Индекс начальной связи для указанной связи.</returns>
211         [MethodImpl(MethodImplOptions.AggressiveInlining)]
212         public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪ link[links.Constants.SourcePart];
213
214         /// <summary>
215         /// Возвращает индекс конечной (Target) связи для указанной связи.
216         /// </summary>
217         /// <param name="links">Хранилище связей.</param>
218         /// <param name="link">Индекс связи.</param>
219         /// <returns>Индекс конечной связи для указанной связи.</returns>
220         [MethodImpl(MethodImplOptions.AggressiveInlining)]
221         public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
            ↪ links.GetLink(link)[links.Constants.TargetPart];
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">Связь представленная списком, состоящим из её адреса и
            ↪ содержимого.</param>
228         /// <returns>Индекс конечной связи для указанной связи.</returns>
229         [MethodImpl(MethodImplOptions.AggressiveInlining)]
230         public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
            ↪ link[links.Constants.TargetPart];
231
232         /// <summary>
233         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
234         /// </summary>
235         /// <param name="links">Хранилище связей.</param>
236         /// <param name="handler">Обработчик каждой подходящей связи.</param>
237         /// <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>
238         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
239         [MethodImpl(MethodImplOptions.AggressiveInlining)]
240         public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
            ↪ handler, params TLink[] restrictions)
241             => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
                ↪ links.Constants.Continue);
242
243         /// <summary>
244         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
245         /// </summary>
246         /// <param name="links">Хранилище связей.</param>
247         /// <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>
248         /// <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>
249         /// <param name="handler">Обработчик каждой подходящей связи.</param>
250         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
```

```
251         [MethodImpl(MethodImplOptions.AggressiveInlining)]
252         public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
            ↪  Func<TLink, bool> handler)
253         {
254             var constants = links.Constants;
255             return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
                ↪  constants.Break, constants.Any, source, target);
256         }
257 
258         /// <summary>
259         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪  (handler) для каждой подходящей связи.
260         /// </summary>
261         /// <param name="links">Хранилище связей.</param>
262         /// <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>
263         /// <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>
264         /// <param name="handler">Обработчик каждой подходящей связи.</param>
265         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪  случае.</returns>
266         [MethodImpl(MethodImplOptions.AggressiveInlining)]
267         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);
268 
269         [MethodImpl(MethodImplOptions.AggressiveInlining)]
270         public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
            ↪  restrictions)
271         {
272             var arraySize = CheckedConverter<TLink,
                ↪  long>.Default.Convert(links.Count(restrictions));
273             if (arraySize > 0)
274             {
275                 var array = new IList<TLink>[arraySize];
276                 var filler = new ArrayFiller<IList<TLink>, TLink>(array,
                    ↪  links.Constants.Continue);
277                 links.Each(filler.AddAndReturnConstant, restrictions);
278                 return array;
279             }
280             else
281             {
282                 return Array.Empty<IList<TLink>>();
283             }
284         }
285 
286         [MethodImpl(MethodImplOptions.AggressiveInlining)]
287         public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
            ↪  restrictions)
288         {
289             var arraySize = CheckedConverter<TLink,
                ↪  long>.Default.Convert(links.Count(restrictions));
290             if (arraySize > 0)
291             {
292                 var array = new TLink[arraySize];
293                 var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
294                 links.Each(filler.AddFirstAndReturnConstant, restrictions);
295                 return array;
296             }
297             else
298             {
299                 return Array.Empty<TLink>();
300             }
301         }
302 
303         /// <summary>
304         /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
            ↪  в хранилище связей.
305         /// </summary>
306         /// <param name="links">Хранилище связей.</param>
307         /// <param name="source">Начало связи.</param>
308         /// <param name="target">Конец связи.</param>
309         /// <returns>Значение, определяющее существует ли связь.</returns>
310         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
311         public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
            => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
            default) > 0;
312
313         #region Ensure
314         // TODO: May be move to EnsureExtensions or make it both there and here
315
316         [MethodImpl(MethodImplOptions.AggressiveInlining)]
317         public static void EnsureLinkExists<TLink>(this ILinks<TLink> links, IList<TLink>
            restrictions)
318         {
319             for (var i = 0; i < restrictions.Count; i++)
320             {
321                 if (!links.Exists(restrictions[i]))
322                 {
323                     throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                    $"sequence[{i}]");
324                 }
325             }
326         }
327
328         [MethodImpl(MethodImplOptions.AggressiveInlining)]
329         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
            reference, string argumentName)
330         {
331             if (links.Constants.IsInternalReference(reference) && !links.Exists(reference))
332             {
333                 throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
334             }
335         }
336
337         [MethodImpl(MethodImplOptions.AggressiveInlining)]
338         public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
            IList<TLink> restrictions, string argumentName)
339         {
340             for (int i = 0; i < restrictions.Count; i++)
341             {
342                 links.EnsureInnerReferenceExists(restrictions[i], argumentName);
343             }
344         }
345
346         [MethodImpl(MethodImplOptions.AggressiveInlining)]
347         public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
            restrictions)
348         {
349             var equalityComparer = EqualityComparer<TLink>.Default;
350             var any = links.Constants.Any;
351             for (var i = 0; i < restrictions.Count; i++)
352             {
353                 if (!equalityComparer.Equals(restrictions[i], any) &&
                    !links.Exists(restrictions[i]))
354                 {
355                     throw new ArgumentLinkDoesNotExistsException<TLink>(restrictions[i],
                    $"sequence[{i}]");
356                 }
357             }
358         }
359
360         [MethodImpl(MethodImplOptions.AggressiveInlining)]
361         public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
            string argumentName)
362         {
363             var equalityComparer = EqualityComparer<TLink>.Default;
364             if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
365             {
366                 throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
367             }
368         }
369
370         [MethodImpl(MethodImplOptions.AggressiveInlining)]
371         public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
            link, string argumentName)
372         {
373             var equalityComparer = EqualityComparer<TLink>.Default;
374             if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
375             {
376                 throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
```

```
            }
        }

        /// <param name="links">Хранилище связей.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
        ↪  TLink target)
        {
            if (links.Exists(source, target))
            {
                throw new LinkWithSameValueAlreadyExistsException();
            }
        }

        /// <param name="links">Хранилище связей.</param>
        public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
        {
            if (links.HasUsages(link))
            {
                throw new ArgumentLinkHasDependenciesException<TLink>(link);
            }
        }

        /// <param name="links">Хранилище связей.</param>
        public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
        ↪  addresses) => links.EnsureCreated(links.Create, addresses);

        /// <param name="links">Хранилище связей.</param>
        public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
        ↪  addresses) => links.EnsureCreated(links.CreatePoint, addresses);

        /// <param name="links">Хранилище связей.</param>
        public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
        ↪  params TLink[] addresses)
        {
            var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
            var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
            var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
            ↪  !links.Exists(x)));
            if (nonExistentAddresses.Count > 0)
            {
                var max = nonExistentAddresses.Max();
                max = uInt64ToAddressConverter.Convert(System.Math.Min(addressToUInt64Converter.
                ↪  Convert(max),
                ↪  addressToUInt64Converter.Convert(links.Constants.InternalReferencesRange.Max
                ↪  imum)));
                var createdLinks = new List<TLink>();
                var equalityComparer = EqualityComparer<TLink>.Default;
                TLink createdLink = creator();
                while (!equalityComparer.Equals(createdLink, max))
                {
                    createdLinks.Add(createdLink);
                }
                for (var i = 0; i < createdLinks.Count; i++)
                {
                    if (!nonExistentAddresses.Contains(createdLinks[i]))
                    {
                        links.Delete(createdLinks[i]);
                    }
                }
            }
        }

        #endregion

        /// <param name="links">Хранилище связей.</param>
        public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
        {
            var constants = links.Constants;
            var values = links.GetLink(link);
            TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link,
            ↪  constants.Any));
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (equalityComparer.Equals(values[constants.SourcePart], link))
            {
                usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
            }
```

```
445             TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
                ↪ link));
446             if (equalityComparer.Equals(values[constants.TargetPart], link))
447             {
448                 usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
449             }
450             return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
451         }
452
453         /// <param name="links">Хранилище связей.</param>
454         [MethodImpl(MethodImplOptions.AggressiveInlining)]
455         public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
                ↪ Comparer<TLink>.Default.Compare(links.CountUsages(link), default) > 0;
456
457         /// <param name="links">Хранилище связей.</param>
458         [MethodImpl(MethodImplOptions.AggressiveInlining)]
459         public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
                ↪ TLink target)
460         {
461             var constants = links.Constants;
462             var values = links.GetLink(link);
463             var equalityComparer = EqualityComparer<TLink>.Default;
464             return equalityComparer.Equals(values[constants.SourcePart], source) &&
                    ↪ equalityComparer.Equals(values[constants.TargetPart], target);
465         }
466
467         /// <summary>
468         /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
469         /// </summary>
470         /// <param name="links">Хранилище связей.</param>
471         /// <param name="source">Индекс связи, которая является началом для искомой
                ↪ связи.</param>
472         /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
473         /// <returns>Индекс искомой связи с указанными Source (началом) и Target
                ↪ (концом).</returns>
474         [MethodImpl(MethodImplOptions.AggressiveInlining)]
475         public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
                ↪ target)
476         {
477             var contants = links.Constants;
478             var setter = new Setter<TLink, TLink>(contants.Continue, contants.Break, default);
479             links.Each(setter.SetFirstAndReturnFalse, contants.Any, source, target);
480             return setter.Result;
481         }
482
483         /// <param name="links">Хранилище связей.</param>
484         [MethodImpl(MethodImplOptions.AggressiveInlining)]
485         public static TLink Create<TLink>(this ILinks<TLink> links) => links.Create(null);
486
487         /// <param name="links">Хранилище связей.</param>
488         [MethodImpl(MethodImplOptions.AggressiveInlining)]
489         public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
490         {
491             var link = links.Create();
492             return links.Update(link, link, link);
493         }
494
495         /// <param name="links">Хранилище связей.</param>
496         [MethodImpl(MethodImplOptions.AggressiveInlining)]
497         public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
                ↪ target) => links.Update(links.Create(), source, target);
498
499         /// <summary>
500         /// Обновляет связь с указанными началом (Source) и концом (Target)
501         /// на связь с указанными началом (NewSource) и концом (NewTarget).
502         /// </summary>
503         /// <param name="links">Хранилище связей.</param>
504         /// <param name="link">Индекс обновляемой связи.</param>
505         /// <param name="newSource">Индекс связи, которая является началом связи, на которую
                ↪ выполняется обновление.</param>
506         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
                ↪ выполняется обновление.</param>
507         /// <returns>Индекс обновлённой связи.</returns>
508         [MethodImpl(MethodImplOptions.AggressiveInlining)]
509         public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
                ↪ TLink newTarget) => links.Update(new LinkAddress<TLink>(link), new Link<TLink>(link,
                ↪ newSource, newTarget));
```

```
510
511         /// <summary>
512         /// Обновляет связь с указанными началом (Source) и концом (Target)
513         /// на связь с указанными началом (NewSource) и концом (NewTarget).
514         /// </summary>
515         /// <param name="links">Хранилище связей.</param>
516         /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
            ↪ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
            ↪ Itself - требование установить ссылку на себя, 1..∞ конкретный адрес другой
            ↪ связи.</param>
517         /// <returns>Индекс обновлённой связи.</returns>
518         [MethodImpl(MethodImplOptions.AggressiveInlining)]
519         public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
520         {
521             if (restrictions.Length == 2)
522             {
523                 return links.MergeAndDelete(restrictions[0], restrictions[1]);
524             }
525             if (restrictions.Length == 4)
526             {
527                 return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
                    ↪ restrictions[2], restrictions[3]);
528             }
529             else
530             {
531                 return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
532             }
533         }
534
535         [MethodImpl(MethodImplOptions.AggressiveInlining)]
536         public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
            ↪ links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
537         {
538             var equalityComparer = EqualityComparer<TLink>.Default;
539             var constants = links.Constants;
540             var restrictionsIndex = restrictions[constants.IndexPart];
541             var substitutionIndex = substitution[constants.IndexPart];
542             if (equalityComparer.Equals(substitutionIndex, default))
543             {
544                 substitutionIndex = restrictionsIndex;
545             }
546             var source = substitution[constants.SourcePart];
547             var target = substitution[constants.TargetPart];
548             source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
549             target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
550             return new Link<TLink>(substitutionIndex, source, target);
551         }
552
553         /// <summary>
554         /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
            ↪ с указанными Source (началом) и Target (концом).
555         /// </summary>
556         /// <param name="links">Хранилище связей.</param>
557         /// <param name="source">Индекс связи, которая является началом на создаваемой
            ↪ связи.</param>
558         /// <param name="target">Индекс связи, которая является концом для создаваемой
            ↪ связи.</param>
559         /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
560         [MethodImpl(MethodImplOptions.AggressiveInlining)]
561         public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target)
562         {
563             var link = links.SearchOrDefault(source, target);
564             if (EqualityComparer<TLink>.Default.Equals(link, default))
565             {
566                 link = links.CreateAndUpdate(source, target);
567             }
568             return link;
569         }
570
571         /// <summary>
572         /// Обновляет связь с указанными началом (Source) и концом (Target)
573         /// на связь с указанными началом (NewSource) и концом (NewTarget).
574         /// </summary>
575         /// <param name="links">Хранилище связей.</param>
576         /// <param name="source">Индекс связи, которая является началом обновляемой
            ↪ связи.</param>
577         /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
```

```
578         /// <param name="newSource">Индекс связи, которая является началом связи, на которую
            ↪ выполняется обновление.</param>
579         /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
            ↪ выполняется обновление.</param>
580         /// <returns>Индекс обновлённой связи.</returns>
581         [MethodImpl(MethodImplOptions.AggressiveInlining)]
582         public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
            ↪ TLink target, TLink newSource, TLink newTarget)
583         {
584             var equalityComparer = EqualityComparer<TLink>.Default;
585             var link = links.SearchOrDefault(source, target);
586             if (equalityComparer.Equals(link, default))
587             {
588                 return links.CreateAndUpdate(newSource, newTarget);
589             }
590             if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
                ↪ target))
591             {
592                 return link;
593             }
594             return links.Update(link, newSource, newTarget);
595         }
596 
597         /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
598         /// <param name="links">Хранилище связей.</param>
599         /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
600         /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
601         [MethodImpl(MethodImplOptions.AggressiveInlining)]
602         public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
            ↪ target)
603         {
604             var link = links.SearchOrDefault(source, target);
605             if (!EqualityComparer<TLink>.Default.Equals(link, default))
606             {
607                 links.Delete(link);
608                 return link;
609             }
610             return default;
611         }
612 
613         /// <summary>Удаляет несколько связей.</summary>
614         /// <param name="links">Хранилище связей.</param>
615         /// <param name="deletedLinks">Список адресов связей к удалению.</param>
616         [MethodImpl(MethodImplOptions.AggressiveInlining)]
617         public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
618         {
619             for (int i = 0; i < deletedLinks.Count; i++)
620             {
621                 links.Delete(deletedLinks[i]);
622             }
623         }
624 
625         /// <remarks>Before execution of this method ensure that deleted link is detached (all
            ↪ values - source and target are reset to null) or it might enter into infinite
            ↪ recursion.</remarks>
626         public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
627         {
628             var anyConstant = links.Constants.Any;
629             var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
630             links.DeleteByQuery(usagesAsSourceQuery);
631             var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
632             links.DeleteByQuery(usagesAsTargetQuery);
633         }
634 
635         public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
636         {
637             var count = CheckedConverter<TLink, long>.Default.Convert(links.Count(query));
638             if (count > 0)
639             {
640                 var queryResult = new TLink[count];
641                 var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
                    ↪ links.Constants.Continue);
642                 links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
643                 for (var i = count - 1; i >= 0; i--)
644                 {
645                     links.Delete(queryResult[i]);
646                 }
647             }
```

```
648         }
649
650         // TODO: Move to Platform.Data
651         public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
652         {
653             var nullConstant = links.Constants.Null;
654             var equalityComparer = EqualityComparer<TLink>.Default;
655             var link = links.GetLink(linkIndex);
656             for (int i = 1; i < link.Count; i++)
657             {
658                 if (!equalityComparer.Equals(link[i], nullConstant))
659                 {
660                     return false;
661                 }
662             }
663             return true;
664         }
665
666         // TODO: Create a universal version of this method in Platform.Data (with using of for
    ↪  loop)
667         public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
668         {
669             var nullConstant = links.Constants.Null;
670             var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
671             links.Update(updateRequest);
672         }
673
674         // TODO: Create a universal version of this method in Platform.Data (with using of for
    ↪  loop)
675         public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
676         {
677             if (!links.AreValuesReset(linkIndex))
678             {
679                 links.ResetValues(linkIndex);
680             }
681         }
682
683         /// <summary>
684         /// Merging two usages graphs, all children of old link moved to be children of new link
    ↪  or deleted.
685         /// </summary>
686         public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
    ↪  TLink newLinkIndex)
687         {
688             var addressToInt64Converter = CheckedConverter<TLink, long>.Default;
689             var equalityComparer = EqualityComparer<TLink>.Default;
690             if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
691             {
692                 var constants = links.Constants;
693                 var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
                    ↪  constants.Any);
694                 var usagesAsSourceCount =
                    ↪  addressToInt64Converter.Convert(links.Count(usagesAsSourceQuery));
695                 var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
                    ↪  oldLinkIndex);
696                 var usagesAsTargetCount =
                    ↪  addressToInt64Converter.Convert(links.Count(usagesAsTargetQuery));
697                 var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
                    ↪  usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
698                 if (!isStandalonePoint)
699                 {
700                     var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
701                     if (totalUsages > 0)
702                     {
703                         var usages = ArrayPool.Allocate<TLink>(totalUsages);
704                         var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
                            ↪  links.Constants.Continue);
705                         var i = 0L;
706                         if (usagesAsSourceCount > 0)
707                         {
708                             links.Each(usagesFiller.AddFirstAndReturnConstant,
                                ↪  usagesAsSourceQuery);
709                             for (; i < usagesAsSourceCount; i++)
710                             {
711                                 var usage = usages[i];
712                                 if (!equalityComparer.Equals(usage, oldLinkIndex))
713                                 {
714                                     links.Update(usage, newLinkIndex, links.GetTarget(usage));
```

```
                                }
                            }
                        }
                        if (usagesAsTargetCount > 0)
                        {
                            links.Each(usagesFiller.AddFirstAndReturnConstant,
                            ↪  usagesAsTargetQuery);
                            for (; i < usages.Length; i++)
                            {
                                var usage = usages[i];
                                if (!equalityComparer.Equals(usage, oldLinkIndex))
                                {
                                    links.Update(usage, links.GetSource(usage), newLinkIndex);
                                }
                            }
                        }
                        ArrayPool.Free(usages);
                    }
                }
            }
            return newLinkIndex;
        }

        /// <summary>
        /// Replace one link with another (replaced link is deleted, children are updated or
        ↪  deleted).
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
        ↪  TLink newLinkIndex)
        {
            var equalityComparer = EqualityComparer<TLink>.Default;
            if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
            {
                links.MergeUsages(oldLinkIndex, newLinkIndex);
                links.Delete(oldLinkIndex);
            }
            return newLinkIndex;
        }

        public static ILinks<TLink>
        ↪  DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
        {
            links = new LinksCascadeUsagesResolver<TLink>(links);
            links = new NonNullContentsLinkDeletionResolver<TLink>(links);
            links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
            return links;
        }
    }
}
```

1.19 ./Platform.Data.Doublets/ISynchronizedLinks.cs

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets
{
    public interface ISynchronizedLinks<TLink> : ISynchronizedLinks<TLink, ILinks<TLink>,
    ↪  LinksConstants<TLink>>, ILinks<TLink>
    {
    }
}
```

1.20 ./Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs

```
using System.Collections.Generic;
using Platform.Incrementers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Incrementers
{
    public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly TLink _frequencyMarker;
        private readonly TLink _unaryOne;
        private readonly IIncrementer<TLink> _unaryNumberIncrementer;

```

```
16          public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
            ↪  IIncrementer<TLink> unaryNumberIncrementer)
17              : base(links)
18          {
19              _frequencyMarker = frequencyMarker;
20              _unaryOne = unaryOne;
21              _unaryNumberIncrementer = unaryNumberIncrementer;
22          }
23
24          public TLink Increment(TLink frequency)
25          {
26              if (_equalityComparer.Equals(frequency, default))
27              {
28                  return Links.GetOrCreate(_unaryOne, _frequencyMarker);
29              }
30              var source = Links.GetSource(frequency);
31              var incrementedSource = _unaryNumberIncrementer.Increment(source);
32              return Links.GetOrCreate(incrementedSource, _frequencyMarker);
33          }
34      }
35  }
```

1.21 ./Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs

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

1.22 ./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 =
            ↪ Default<LinksConstants<TLink>>.Instance;
22          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
23
24          private const int Length = 3;
25
26          public readonly TLink Index;
27          public readonly TLink Source;
28          public readonly TLink Target;
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public Link(params TLink[] values) => SetValues(values, out Index, out Source, out
            ↪ Target);
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public Link(IList<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 IList<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(IList<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;
```

```
                source = default;
                target = default;
                break;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => (Index, Source, Target).GetHashCode();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
                            && _equalityComparer.Equals(Source, _constants.Null)
                            && _equalityComparer.Equals(Target, _constants.Null);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object other) => other is Link<TLink> &&
        ↪  Equals((Link<TLink>)other);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
                                              && _equalityComparer.Equals(Source, other.Source)
                                              && _equalityComparer.Equals(Target, other.Target);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string ToString(TLink index, TLink source, TLink target) => $"({index}:
        ↪  {source}->{target})";

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string ToString(TLink source, TLink target) => $"({source}->{target})";

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator TLink[](Link<TLink> link) => link.ToArray();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator Link<TLink>(TLink[] linkArray) => new
        ↪  Link<TLink>(linkArray);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
        ↪  ToString(Source, Target) : ToString(Index, Source, Target);

        #region IList

        public int Count => Length;

        public bool IsReadOnly => true;

        public TLink this[int index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
                ↪  nameof(index));
                if (index == _constants.IndexPart)
                {
                    return Index;
                }
                if (index == _constants.SourcePart)
                {
                    return Source;
                }
                if (index == _constants.TargetPart)
                {
                    return Target;
                }
                throw new NotSupportedException(); // Impossible path due to
                ↪  Ensure.ArgumentInRange
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => throw new NotSupportedException();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerator<TLink> GetEnumerator()
        {
```

```
            yield return Index;
            yield return Source;
            yield return Target;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(TLink item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Clear() => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Contains(TLink item) => IndexOf(item) >= 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void CopyTo(TLink[] array, int arrayIndex)
        {
            Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
            Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
            ↪  nameof(arrayIndex));
            if (arrayIndex + Length > array.Length)
            {
                throw new InvalidOperationException();
            }
            array[arrayIndex++] = Index;
            array[arrayIndex++] = Source;
            array[arrayIndex] = Target;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int IndexOf(TLink item)
        {
            if (_equalityComparer.Equals(Index, item))
            {
                return _constants.IndexPart;
            }
            if (_equalityComparer.Equals(Source, item))
            {
                return _constants.SourcePart;
            }
            if (_equalityComparer.Equals(Target, item))
            {
                return _constants.TargetPart;
            }
            return -1;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Insert(int index, TLink item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void RemoveAt(int index) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator ==(Link<TLink> left, Link<TLink> right) =>
        ↪  left.Equals(right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator !=(Link<TLink> left, Link<TLink> right) => !(left == right);

        #endregion
    }
}
```

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

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets
{
    public static class LinkExtensions
    {
        public static bool IsFullPoint<TLink>(this Link<TLink> link) =>
        ↪  Point<TLink>.IsFullPoint(link);
        public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
        ↪  Point<TLink>.IsPartialPoint(link);
    }
```

```
}
```

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

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets
{
    public abstract class LinksOperatorBase<TLink>
    {
        public ILinks<TLink> Links { get; }
        protected LinksOperatorBase(ILinks<TLink> links) => Links = links;
    }
}
```

1.25 ./Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs

```
using System.Collections.Generic;
using Platform.Reflection;
using Platform.Converters;
using Platform.Numbers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Numbers.Unary
{
    public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;
        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;

        public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
        ↪ powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
        ↪ powerOf2ToUnaryNumberConverter;

        public TLink Convert(TLink number)
        {
            var nullConstant = Links.Constants.Null;
            var target = nullConstant;
            for (var i = 0; !_equalityComparer.Equals(number, _zero) && i <
            ↪ NumericType<TLink>.BitsSize; i++)
            {
                if (_equalityComparer.Equals(Bit.And(number, _one), _one))
                {
                    target = _equalityComparer.Equals(target, nullConstant)
                        ? _powerOf2ToUnaryNumberConverter.Convert(i)
                        : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
                }
                number = Bit.ShiftRight(number, 1);
            }
            return target;
        }
    }
}
```

1.26 ./Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConveter.cs

```
using System;
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Converters;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Numbers.Unary
{
    public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<Doublet<TLink>, TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;

        private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
        private readonly IConverter<TLink> _unaryNumberToAddressConverter;

        public LinkToItsFrequencyNumberConveter(
            ILinks<TLink> links,
            IProperty<TLink, TLink> frequencyPropertyOperator,
```

```
            IConverter<TLink> unaryNumberToAddressConverter)
            : base(links)
        {
            _frequencyPropertyOperator = frequencyPropertyOperator;
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
        }

        public TLink Convert(Doublet<TLink> doublet)
        {
            var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
            if (_equalityComparer.Equals(link, default))
            {
                throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
            }
            var frequency = _frequencyPropertyOperator.Get(link);
            if (_equalityComparer.Equals(frequency, default))
            {
                return default;
            }
            var frequencyNumber = Links.GetSource(frequency);
            return _unaryNumberToAddressConverter.Convert(frequencyNumber);
        }
    }
}
```

1.27 ./Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs

```
using System.Collections.Generic;
using Platform.Exceptions;
using Platform.Ranges;
using Platform.Converters;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Numbers.Unary
{
    public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<int, TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly TLink[] _unaryNumberPowersOf2;

        public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
        {
            _unaryNumberPowersOf2 = new TLink[64];
            _unaryNumberPowersOf2[0] = one;
        }

        public TLink Convert(int power)
        {
            Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
            ↪  - 1), nameof(power));
            if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
            {
                return _unaryNumberPowersOf2[power];
            }
            var previousPowerOf2 = Convert(power - 1);
            var powerOf2 = Links.GetOrCreate(previousPowerOf2, previousPowerOf2);
            _unaryNumberPowersOf2[power] = powerOf2;
            return powerOf2;
        }
    }
}
```

1.28 ./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Converters;
using Platform.Numbers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Numbers.Unary
{
    public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
```

```
13          private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
            ↪  UncheckedConverter<TLink, ulong>.Default;
14          private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
            ↪  UncheckedConverter<ulong, TLink>.Default;
15          private static readonly TLink _zero = default;
16          private static readonly TLink _one = Arithmetic.Increment(_zero);
17
18          private Dictionary<TLink, TLink> _unaryToUInt64;
19          private readonly TLink _unaryOne;
20
21          public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
22              : base(links)
23          {
24              _unaryOne = unaryOne;
25              InitUnaryToUInt64();
26          }
27
28          private void InitUnaryToUInt64()
29          {
30              _unaryToUInt64 = new Dictionary<TLink, TLink>
31              {
32                  { _unaryOne, _one }
33              };
34              var unary = _unaryOne;
35              var number = _one;
36              for (var i = 1; i < 64; i++)
37              {
38                  unary = Links.GetOrCreate(unary, unary);
39                  number = Double(number);
40                  _unaryToUInt64.Add(unary, number);
41              }
42          }
43
44          public TLink Convert(TLink unaryNumber)
45          {
46              if (_equalityComparer.Equals(unaryNumber, default))
47              {
48                  return default;
49              }
50              if (_equalityComparer.Equals(unaryNumber, _unaryOne))
51              {
52                  return _one;
53              }
54              var source = Links.GetSource(unaryNumber);
55              var target = Links.GetTarget(unaryNumber);
56              if (_equalityComparer.Equals(source, target))
57              {
58                  return _unaryToUInt64[unaryNumber];
59              }
60              else
61              {
62                  var result = _unaryToUInt64[source];
63                  TLink lastValue;
64                  while (!_unaryToUInt64.TryGetValue(target, out lastValue))
65                  {
66                      source = Links.GetSource(target);
67                      result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
68                      target = Links.GetTarget(target);
69                  }
70                  result = Arithmetic<TLink>.Add(result, lastValue);
71                  return result;
72              }
73          }
74
75          [MethodImpl(MethodImplOptions.AggressiveInlining)]
76          private static TLink Double(TLink number) =>
            ↪  _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
77      }
78  }
```

1.29 ./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
```

```
{
    public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;
        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;

        public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
        ↪ TLink> powerOf2ToUnaryNumberConverter)
            : base(links)
        {
            _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
            for (int i = 0; i < NumericType<TLink>.BitsSize; i++)
            {
                _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
            }
        }

        public TLink Convert(TLink sourceNumber)
        {
            var nullConstant = Links.Constants.Null;
            var source = sourceNumber;
            var target = nullConstant;
            if (!_equalityComparer.Equals(source, nullConstant))
            {
                while (true)
                {
                    if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
                    {
                        SetBit(ref target, powerOf2Index);
                        break;
                    }
                    else
                    {
                        powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
                        SetBit(ref target, powerOf2Index);
                        source = Links.GetTarget(source);
                    }
                }
            }
            return target;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void SetBit(ref TLink target, int powerOf2Index) => target =
        ↪ Bit.Or(target, Bit.ShiftLeft(_one, powerOf2Index));
    }
}
```

1.30 ./Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs

```
using System.Linq;
using System.Collections.Generic;
using Platform.Interfaces;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.PropertyOperators
{
    public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>, IProperties<TLink, TLink,
    ↪ TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;

        public PropertiesOperator(ILinks<TLink> links) : base(links) { }

        public TLink GetValue(TLink @object, TLink property)
        {
            var objectProperty = Links.SearchOrDefault(@object, property);
            if (_equalityComparer.Equals(objectProperty, default))
            {
                return default;
            }
            var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
            if (valueLink == null)
            {
```

```
                return default;
            }
            return Links.GetTarget(valueLink[Links.Constants.IndexPart]);
        }

        public void SetValue(TLink @object, TLink property, TLink value)
        {
            var objectProperty = Links.GetOrCreate(@object, property);
            Links.DeleteMany(Links.AllIndices(Links.Constants.Any, objectProperty));
            Links.GetOrCreate(objectProperty, value);
        }
    }
}
```

1.31 ./Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs

```
using System.Collections.Generic;
using Platform.Interfaces;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.PropertyOperators
{
    public class PropertyOperator<TLink> : LinksOperatorBase<TLink>, IProperty<TLink, TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;

        private readonly TLink _propertyMarker;
        private readonly TLink _propertyValueMarker;

        public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
        ↪ propertyValueMarker) : base(links)
        {
            _propertyMarker = propertyMarker;
            _propertyValueMarker = propertyValueMarker;
        }

        public TLink Get(TLink link)
        {
            var property = Links.SearchOrDefault(link, _propertyMarker);
            var container = GetContainer(property);
            var value = GetValue(container);
            return value;
        }

        private TLink GetContainer(TLink property)
        {
            var valueContainer = default(TLink);
            if (_equalityComparer.Equals(property, default))
            {
                return valueContainer;
            }
            var constants = Links.Constants;
            var countinueConstant = constants.Continue;
            var breakConstant = constants.Break;
            var anyConstant = constants.Any;
            var query = new Link<TLink>(anyConstant, property, anyConstant);
            Links.Each(candidate =>
            {
                var candidateTarget = Links.GetTarget(candidate);
                var valueTarget = Links.GetTarget(candidateTarget);
                if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
                {
                    valueContainer = Links.GetIndex(candidate);
                    return breakConstant;
                }
                return countinueConstant;
            }, query);
            return valueContainer;
        }

        private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
        ↪ ? default : Links.GetTarget(container);

        public void Set(TLink link, TLink value)
        {
            var property = Links.GetOrCreate(link, _propertyMarker);
            var container = GetContainer(property);
            if (_equalityComparer.Equals(container, default))
            {
```

```
                Links.GetOrCreate(property, value);
            }
            else
            {
                Links.Update(container, property, value);
            }
        }
    }
}
```

1.32 ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksAvlBalancedTreeMethodsBase.cs

```
using System;
using System.Text;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Methods.Trees;
using Platform.Converters;
using Platform.Numbers;
using static System.Runtime.CompilerServices.Unsafe;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
{
    public unsafe abstract class LinksAvlBalancedTreeMethodsBase<TLink> :
    ↪ SizedAndThreadedAVLBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
        ↪ UncheckedConverter<TLink, long>.Default;
        private static readonly UncheckedConverter<TLink, int> _addressToInt32Converter =
        ↪ UncheckedConverter<TLink, int>.Default;
        private static readonly UncheckedConverter<bool, TLink> _boolToAddressConverter =
        ↪ UncheckedConverter<bool, TLink>.Default;
        private static readonly UncheckedConverter<int, TLink> _int32ToAddressConverter =
        ↪ UncheckedConverter<int, TLink>.Default;

        protected readonly TLink Break;
        protected readonly TLink Continue;
        protected readonly byte* Links;
        protected readonly byte* Header;

        protected LinksAvlBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links,
        ↪ byte* header)
        {
            Links = links;
            Header = header;
            Break = constants.Break;
            Continue = constants.Continue;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLink GetTreeRoot();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLink GetBasePartValue(TLink link);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink
        ↪ rootSource, TLink rootTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink
        ↪ rootSource, TLink rootTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
        ↪ AsRef<LinksHeader<TLink>>(Header);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
        ↪ AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes *
        ↪ _addressToInt64Converter.Convert(link)));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual IList<TLink> GetLinkValues(TLink linkIndex)
        {
            ref var link = ref GetLinkReference(linkIndex);
            return new Link<TLink>(linkIndex, link.Source, link.Target);
        }

```

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

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

```
                {
                    root = GetRightOrDefault(root);
                }
                else // node.Key == root.Key
                {
                    return root;
                }
            }
            return Zero;
        }

        // TODO: Return indices range instead of references count
        public TLink CountUsages(TLink link)
        {
            var root = GetTreeRoot();
            var total = GetSize(root);
            var totalRightIgnore = Zero;
            while (!EqualToZero(root))
            {
                var @base = GetBasePartValue(root);
                if (LessOrEqualThan(@base, link))
                {
                    root = GetRightOrDefault(root);
                }
                else
                {
                    totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
                    root = GetLeftOrDefault(root);
                }
            }
            root = GetTreeRoot();
            var totalLeftIgnore = Zero;
            while (!EqualToZero(root))
            {
                var @base = GetBasePartValue(root);
                if (GreaterOrEqualThan(@base, link))
                {
                    root = GetLeftOrDefault(root);
                }
                else
                {
                    totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));

                    root = GetRightOrDefault(root);
                }
            }
            return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
        }

        public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
        {
            var root = GetTreeRoot();
            if (EqualToZero(root))
            {
                return Continue;
            }
            TLink first = Zero, current = root;
            while (!EqualToZero(current))
            {
                var @base = GetBasePartValue(current);
                if (GreaterOrEqualThan(@base, link))
                {
                    if (AreEqual(@base, link))
                    {
                        first = current;
                    }
                    current = GetLeftOrDefault(current);
                }
                else
                {
                    current = GetRightOrDefault(current);
                }
            }
            if (!EqualToZero(first))
            {
                current = first;
                while (true)
                {
                    if (AreEqual(handler(GetLinkValues(current)), Break))
```

```
                    {
                        return Break;
                    }
                    current = GetNext(current);
                    if (EqualToZero(current) || !AreEqual(GetBasePartValue(current), link))
                    {
                        break;
                    }
                }
            }
            return Continue;
        }

        protected override void PrintNodeValue(TLink node, StringBuilder sb)
        {
            ref var link = ref GetLinkReference(node);
            sb.Append(' ');
            sb.Append(link.Source);
            sb.Append('-');
            sb.Append('>');
            sb.Append(link.Target);
        }
    }
}
```

1.33 ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSizeBalancedTreeMethodsBase.cs

```
using System;
using System.Text;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Methods.Trees;
using Platform.Converters;
using static System.Runtime.CompilerServices.Unsafe;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
{
    public unsafe abstract class LinksSizeBalancedTreeMethodsBase<TLink> : SizeBalancedTreeMethods<TLink>, ILinksTreeMethods<TLink>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter = UncheckedConverter<TLink, long>.Default;

        protected readonly TLink Break;
        protected readonly TLink Continue;
        protected readonly byte* Links;
        protected readonly byte* Header;

        protected LinksSizeBalancedTreeMethodsBase(LinksConstants<TLink> constants, byte* links, byte* header)
        {
            Links = links;
            Header = header;
            Break = constants.Break;
            Continue = constants.Continue;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLink GetTreeRoot();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLink GetBasePartValue(TLink link);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheRightOfSecond(TLink source, TLink target, TLink rootSource, TLink rootTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract bool FirstIsToTheLeftOfSecond(TLink source, TLink target, TLink rootSource, TLink rootTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref AsRef<LinksHeader<TLink>>(Header);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref AsRef<RawLink<TLink>>(Links + (RawLink<TLink>.SizeInBytes * _addressToInt64Converter.Convert(link)));
```

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

```
                {
                    root = GetRightOrDefault(root);
                }
                else // node.Key == root.Key
                {
                    return root;
                }
            }
            return Zero;
        }

        // TODO: Return indices range instead of references count
        public TLink CountUsages(TLink link)
        {
            var root = GetTreeRoot();
            var total = GetSize(root);
            var totalRightIgnore = Zero;
            while (!EqualToZero(root))
            {
                var @base = GetBasePartValue(root);
                if (LessOrEqualThan(@base, link))
                {
                    root = GetRightOrDefault(root);
                }
                else
                {
                    totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
                    root = GetLeftOrDefault(root);
                }
            }
            root = GetTreeRoot();
            var totalLeftIgnore = Zero;
            while (!EqualToZero(root))
            {
                var @base = GetBasePartValue(root);
                if (GreaterOrEqualThan(@base, link))
                {
                    root = GetLeftOrDefault(root);
                }
                else
                {
                    totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));

                    root = GetRightOrDefault(root);
                }
            }
            return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink EachUsage(TLink @base, Func<IList<TLink>, TLink> handler) =>
        ↪  EachUsageCore(@base, GetTreeRoot(), handler);

        // TODO: 1. Move target, handler to separate object. 2. Use stack or walker 3. Use
        ↪  low-level MSIL stack.
        private TLink EachUsageCore(TLink @base, TLink link, Func<IList<TLink>, TLink> handler)
        {
            var @continue = Continue;
            if (EqualToZero(link))
            {
                return @continue;
            }
            var linkBasePart = GetBasePartValue(link);
            var @break = Break;
            if (GreaterThan(linkBasePart, @base))
            {
                if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
                {
                    return @break;
                }
            }
            else if (LessThan(linkBasePart, @base))
            {
                if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
                {
                    return @break;
                }
            }
            else //if (linkBasePart == @base)
```

```
            {
                if (AreEqual(handler(GetLinkValues(link)), @break))
                {
                    return @break;
                }
                if (AreEqual(EachUsageCore(@base, GetLeftOrDefault(link), handler), @break))
                {
                    return @break;
                }
                if (AreEqual(EachUsageCore(@base, GetRightOrDefault(link), handler), @break))
                {
                    return @break;
                }
            }
            return @continue;
        }

        protected override void PrintNodeValue(TLink node, StringBuilder sb)
        {
            ref var link = ref GetLinkReference(node);
            sb.Append(' ');
            sb.Append(link.Source);
            sb.Append('-');
            sb.Append('>');
            sb.Append(link.Target);
        }
    }
}
```

1.34 ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesAvlBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
{
    public unsafe class LinksSourcesAvlBalancedTreeMethods<TLink> :
    ↪ LinksAvlBalancedTreeMethodsBase<TLink>
    {
        public LinksSourcesAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
        ↪ byte* header) : base(constants, links, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected unsafe override ref TLink GetLeftReference(TLink node) => ref
        ↪ GetLinkReference(node).LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected unsafe override ref TLink GetRightReference(TLink node) => ref
        ↪ GetLinkReference(node).RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLink node, TLink left) =>
        ↪ GetLinkReference(node).LeftAsSource = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
        ↪ GetLinkReference(node).RightAsSource = right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) =>
        ↪ GetSizeValue(GetLinkReference(node).SizeAsSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
        ↪ GetLinkReference(node).SizeAsSource, size);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetLeftIsChild(TLink node) =>
        ↪ GetLeftIsChildValue(GetLinkReference(node).SizeAsSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeftIsChild(TLink node, bool value) =>
        ↪ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsSource, value);
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetRightIsChild(TLink node) =>
        ↪ GetRightIsChildValue(GetLinkReference(node).SizeAsSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRightIsChild(TLink node, bool value) =>
        ↪ SetRightIsChildValue(ref GetLinkReference(node).SizeAsSource, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override sbyte GetBalance(TLink node) =>
        ↪ GetBalanceValue(GetLinkReference(node).SizeAsSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
        ↪ GetLinkReference(node).SizeAsSource, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↪ TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
        ↪ (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
        ↪ (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkReference(node);
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

1.35 ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksSourcesSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
{
    public unsafe class LinksSourcesSizeBalancedTreeMethods<TLink> :
    ↪ LinksSizeBalancedTreeMethodsBase<TLink>
    {
        public LinksSourcesSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
        ↪ byte* header) : base(constants, links, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected unsafe override ref TLink GetLeftReference(TLink node) => ref
        ↪ GetLinkReference(node).LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected unsafe override ref TLink GetRightReference(TLink node) => ref
        ↪ GetLinkReference(node).RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLink node, TLink left) =>
        ↪ GetLinkReference(node).LeftAsSource = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
        ↪ GetLinkReference(node).RightAsSource = right;

```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) =>
        ↪  GetLinkReference(node).SizeAsSource = size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsSource;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Source;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↪  TLink secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
        ↪  (AreEqual(firstSource, secondSource) && LessThan(firstTarget, secondTarget));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
        ↪  (AreEqual(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkReference(node);
            link.LeftAsSource = Zero;
            link.RightAsSource = Zero;
            link.SizeAsSource = Zero;
        }
    }
}
```

1.36 ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsAvlBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
{
    public unsafe class LinksTargetsAvlBalancedTreeMethods<TLink> :
    ↪  LinksAvlBalancedTreeMethodsBase<TLink>
    {
        public LinksTargetsAvlBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
        ↪  byte* header) : base(constants, links, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected unsafe override ref TLink GetLeftReference(TLink node) => ref
        ↪  GetLinkReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected unsafe override ref TLink GetRightReference(TLink node) => ref
        ↪  GetLinkReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeft(TLink node, TLink left) =>
        ↪  GetLinkReference(node).LeftAsTarget = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
        ↪  GetLinkReference(node).RightAsTarget = right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) =>
        ↪  GetSizeValue(GetLinkReference(node).SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) => SetSizeValue(ref
        ↪  GetLinkReference(node).SizeAsTarget, size);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected override bool GetLeftIsChild(TLink node) =>
        ↪ GetLeftIsChildValue(GetLinkReference(node).SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeftIsChild(TLink node, bool value) =>
        ↪ SetLeftIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetRightIsChild(TLink node) =>
        ↪ GetRightIsChildValue(GetLinkReference(node).SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRightIsChild(TLink node, bool value) =>
        ↪ SetRightIsChildValue(ref GetLinkReference(node).SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override sbyte GetBalance(TLink node) =>
        ↪ GetBalanceValue(GetLinkReference(node).SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetBalance(TLink node, sbyte value) => SetBalanceValue(ref
        ↪ GetLinkReference(node).SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↪ TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
        ↪ (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↪ TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
        ↪ (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkReference(node);
            link.LeftAsTarget = Zero;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }
    }
}
```

1.37 ./Platform.Data.Doublets/ResizableDirectMemory/Generic/LinksTargetsSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
{
    public unsafe class LinksTargetsSizeBalancedTreeMethods<TLink> :
    ↪ LinksSizeBalancedTreeMethodsBase<TLink>
    {
        public LinksTargetsSizeBalancedTreeMethods(LinksConstants<TLink> constants, byte* links,
        ↪ byte* header) : base(constants, links, header) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected unsafe override ref TLink GetLeftReference(TLink node) => ref
        ↪ GetLinkReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected unsafe override ref TLink GetRightReference(TLink node) => ref
        ↪ GetLinkReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLeft(TLink node) => GetLinkReference(node).LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetRight(TLink node) => GetLinkReference(node).RightAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected override void SetLeft(TLink node, TLink left) =>
        ↪  GetLinkReference(node).LeftAsTarget = left;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRight(TLink node, TLink right) =>
        ↪  GetLinkReference(node).RightAsTarget = right;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize(TLink node) => GetLinkReference(node).SizeAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink node, TLink size) =>
        ↪  GetLinkReference(node).SizeAsTarget = size;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetTreeRoot() => GetHeaderReference().FirstAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetBasePartValue(TLink link) => GetLinkReference(link).Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget,
        ↪  TLink secondSource, TLink secondTarget) => LessThan(firstTarget, secondTarget) ||
        ↪  (AreEqual(firstTarget, secondTarget) && LessThan(firstSource, secondSource));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget,
        ↪  TLink secondSource, TLink secondTarget) => GreaterThan(firstTarget, secondTarget) ||
        ↪  (AreEqual(firstTarget, secondTarget) && GreaterThan(firstSource, secondSource));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(TLink node)
        {
            ref var link = ref GetLinkReference(node);
            link.LeftAsTarget = Zero;
            link.RightAsTarget = Zero;
            link.SizeAsTarget = Zero;
        }
    }
}
```

1.38 ./Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinks.cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Singletons;
using Platform.Memory;
using static System.Runtime.CompilerServices.Unsafe;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.ResizableDirectMemory.Generic
{
    public unsafe partial class ResizableDirectMemoryLinks<TLink> :
    ↪  ResizableDirectMemoryLinksBase<TLink>
    {
        private readonly Func<ILinksTreeMethods<TLink>> _createSourceTreeMethods;
        private readonly Func<ILinksTreeMethods<TLink>> _createTargetTreeMethods;
        private byte* _header;
        private byte* _links;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ResizableDirectMemoryLinks(string address) : this(address, DefaultLinksSizeStep)
        ↪  { }

        /// <summary>
        /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
        ↪  минимальным шагом расширения базы данных.
        /// </summary>
        /// <param name="address">Полный пусть к файлу базы данных.</param>
        /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
        ↪  байтах.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ResizableDirectMemoryLinks(string address, long memoryReservationStep) : this(new
        ↪  FileMappedResizableDirectMemory(address, memoryReservationStep),
        ↪  memoryReservationStep) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
        ↪  DefaultLinksSizeStep) { }
```

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

1.39 ./Platform.Data.Doublets/ResizableDirectMemory/Generic/ResizableDirectMemoryLinksBase.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.ResizableDirectMemory.Generic
14  {
15      public abstract class ResizableDirectMemoryLinksBase<TLink> : DisposableBase, ILinks<TLink>
16      {
17          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
18          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
19          private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            ↪  UncheckedConverter<TLink, long>.Default;
```

```
20          private static readonly UncheckedConverter<long, TLink> _int64ToAddressConverter =
            ↪  UncheckedConverter<long, TLink>.Default;
21
22          private static readonly TLink _zero = default;
23          private static readonly TLink _one = Arithmetic.Increment(_zero);
24
25          /// <summary>Возвращает размер одной связи в байтах.</summary>
26          /// <remarks>
27          /// Используется только во вне класса, не рекомедуется использовать внутри.
28          /// Так как во вне не обязательно будет доступен unsafe C#.
29          /// </remarks>
30          public static readonly long LinkSizeInBytes = RawLink<TLink>.SizeInBytes;
31
32          public static readonly long LinkHeaderSizeInBytes = LinksHeader<TLink>.SizeInBytes;
33
34          public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
35
36          protected readonly IResizableDirectMemory _memory;
37          protected readonly long _memoryReservationStep;
38
39          protected ILinksTreeMethods<TLink> TargetsTreeMethods;
40          protected ILinksTreeMethods<TLink> SourcesTreeMethods;
41          // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
            ↪  нужно использовать не список а дерево, так как так можно быстрее проверить на
            ↪  наличие связи внутри
42          protected ILinksListMethods<TLink> UnusedLinksListMethods;
43
44          /// <summary>
45          /// Возвращает общее число связей находящихся в хранилище.
46          /// </summary>
47          protected virtual TLink Total
48          {
49              get
50              {
51                  ref var header = ref GetHeaderReference();
52                  return Subtract(header.AllocatedLinks, header.FreeLinks);
53              }
54          }
55
56          public virtual LinksConstants<TLink> Constants { get; }
57
58          [MethodImpl(MethodImplOptions.AggressiveInlining)]
59          protected ResizableDirectMemoryLinksBase(IResizableDirectMemory memory, long
            ↪  memoryReservationStep, LinksConstants<TLink> constants)
60          {
61              _memory = memory;
62              _memoryReservationStep = memoryReservationStep;
63              Constants = constants;
64          }
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          protected ResizableDirectMemoryLinksBase(IResizableDirectMemory memory, long
            ↪  memoryReservationStep) : this(memory, memoryReservationStep,
            ↪  Default<LinksConstants<TLink>>.Instance) { }
68
69          protected virtual void Init(IResizableDirectMemory memory, long memoryReservationStep)
70          {
71              if (memory.ReservedCapacity < memoryReservationStep)
72              {
73                  memory.ReservedCapacity = memoryReservationStep;
74              }
75              SetPointers(_memory);
76              ref var header = ref GetHeaderReference();
77              // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
78              _memory.UsedCapacity = (ConvertToInt64(header.AllocatedLinks) * LinkSizeInBytes) +
                ↪  LinkHeaderSizeInBytes;
79              // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
80              header.ReservedLinks = ConvertToAddress((_memory.ReservedCapacity -
                ↪  LinkHeaderSizeInBytes) / LinkSizeInBytes);
81          }
82
83          [MethodImpl(MethodImplOptions.AggressiveInlining)]
84          public virtual TLink Count(IList<TLink> restrictions)
85          {
86              // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
87              if (restrictions.Count == 0)
88              {
89                  return Total;
90              }
91              var constants = Constants;
```

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

```
                    {
                        if (AreEqual(storedLinkValue.Source, source) &&
                        ↪  AreEqual(storedLinkValue.Target, target))
                        {
                            return GetOne();
                        }
                        return GetZero();
                    }
                    var value = default(TLink);
                    if (AreEqual(source, any))
                    {
                        value = target;
                    }
                    if (AreEqual(target, any))
                    {
                        value = source;
                    }
                    if (AreEqual(storedLinkValue.Source, value) ||
                    ↪  AreEqual(storedLinkValue.Target, value))
                    {
                        return GetOne();
                    }
                    return GetZero();
                }
            }
            throw new NotSupportedException("Другие размеры и способы ограничений не
            ↪  поддерживаются.");
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        {
            var constants = Constants;
            var @break = constants.Break;
            if (restrictions.Count == 0)
            {
                for (var link = GetOne(); LessOrEqualThan(link,
                ↪  GetHeaderReference().AllocatedLinks); link = Increment(link))
                {
                    if (Exists(link) && AreEqual(handler(GetLinkStruct(link)), @break))
                    {
                        return @break;
                    }
                }
                return @break;
            }
            var @continue = constants.Continue;
            var any = constants.Any;
            var index = restrictions[constants.IndexPart];
            if (restrictions.Count == 1)
            {
                if (AreEqual(index, any))
                {
                    return Each(handler, GetEmptyList());
                }
                if (!Exists(index))
                {
                    return @continue;
                }
                return handler(GetLinkStruct(index));
            }
            if (restrictions.Count == 2)
            {
                var value = restrictions[1];
                if (AreEqual(index, any))
                {
                    if (AreEqual(value, any))
                    {
                        return Each(handler, GetEmptyList());
                    }
                    if (AreEqual(Each(handler, new Link<TLink>(index, value, any)), @break))
                    {
                        return @break;
                    }
                    return Each(handler, new Link<TLink>(index, any, value));
                }
                else
                {
```

```
                    if (!Exists(index))
                    {
                        return @continue;
                    }
                    if (AreEqual(value, any))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    ref var storedLinkValue = ref GetLinkReference(index);
                    if (AreEqual(storedLinkValue.Source, value) ||
                        AreEqual(storedLinkValue.Target, value))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return @continue;
                }
            }
            if (restrictions.Count == 3)
            {
                var source = restrictions[constants.SourcePart];
                var target = restrictions[constants.TargetPart];
                if (AreEqual(index, any))
                {
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return Each(handler, GetEmptyList());
                    }
                    else if (AreEqual(source, any))
                    {
                        return TargetsTreeMethods.EachUsage(target, handler);
                    }
                    else if (AreEqual(target, any))
                    {
                        return SourcesTreeMethods.EachUsage(source, handler);
                    }
                    else //if(source != Any && target != Any)
                    {
                        var link = SourcesTreeMethods.Search(source, target);
                        return AreEqual(link, constants.Null) ? @continue :
                        ↪  handler(GetLinkStruct(link));
                    }
                }
                else
                {
                    if (!Exists(index))
                    {
                        return @continue;
                    }
                    if (AreEqual(source, any) && AreEqual(target, any))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    ref var storedLinkValue = ref GetLinkReference(index);
                    if (!AreEqual(source, any) && !AreEqual(target, any))
                    {
                        if (AreEqual(storedLinkValue.Source, source) &&
                            AreEqual(storedLinkValue.Target, target))
                        {
                            return handler(GetLinkStruct(index));
                        }
                        return @continue;
                    }
                    var value = default(TLink);
                    if (AreEqual(source, any))
                    {
                        value = target;
                    }
                    if (AreEqual(target, any))
                    {
                        value = source;
                    }
                    if (AreEqual(storedLinkValue.Source, value) ||
                        AreEqual(storedLinkValue.Target, value))
                    {
                        return handler(GetLinkStruct(index));
                    }
                    return @continue;
                }
```

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

```
393             var link = restrictions[Constants.IndexPart];
394             if (LessThan(link, header.AllocatedLinks))
395             {
396                 UnusedLinksListMethods.AttachAsFirst(link);
397             }
398             else if (AreEqual(link, header.AllocatedLinks))
399             {
400                 header.AllocatedLinks = Decrement(header.AllocatedLinks);
401                 _memory.UsedCapacity -= LinkSizeInBytes;
402                 // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
                    ↪  пока не дойдём до первой существующей связи
403                 // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
404                 while (GreaterThan(header.AllocatedLinks, GetZero()) &&
                        ↪  IsUnusedLink(header.AllocatedLinks))
405                 {
406                     UnusedLinksListMethods.Detach(header.AllocatedLinks);
407                     header.AllocatedLinks = Decrement(header.AllocatedLinks);
408                     _memory.UsedCapacity -= LinkSizeInBytes;
409                 }
410             }
411         }
412 
413         [MethodImpl(MethodImplOptions.AggressiveInlining)]
414         public IList<TLink> GetLinkStruct(TLink linkIndex)
415         {
416             ref var link = ref GetLinkReference(linkIndex);
417             return new Link<TLink>(linkIndex, link.Source, link.Target);
418         }
419 
420         /// <remarks>
421         /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
                ↪  адрес реально поменялся
422         ///
423         /// Указатель this.links может быть в том же месте,
424         /// так как 0-я связь не используется и имеет такой же размер как Header,
425         /// поэтому header размещается в том же месте, что и 0-я связь
426         /// </remarks>
427         [MethodImpl(MethodImplOptions.AggressiveInlining)]
428         protected abstract void SetPointers(IResizableDirectMemory memory);
429 
430         [MethodImpl(MethodImplOptions.AggressiveInlining)]
431         protected virtual void ResetPointers()
432         {
433             SourcesTreeMethods = null;
434             TargetsTreeMethods = null;
435             UnusedLinksListMethods = null;
436         }
437 
438         [MethodImpl(MethodImplOptions.AggressiveInlining)]
439         protected abstract ref LinksHeader<TLink> GetHeaderReference();
440 
441         [MethodImpl(MethodImplOptions.AggressiveInlining)]
442         protected abstract ref RawLink<TLink> GetLinkReference(TLink linkIndex);
443 
444         [MethodImpl(MethodImplOptions.AggressiveInlining)]
445         protected virtual bool Exists(TLink link)
446             => GreaterOrEqualThan(link, Constants.InternalReferencesRange.Minimum)
447             && LessOrEqualThan(link, GetHeaderReference().AllocatedLinks)
448             && !IsUnusedLink(link);
449 
450         [MethodImpl(MethodImplOptions.AggressiveInlining)]
451         protected virtual bool IsUnusedLink(TLink linkIndex)
452         {
453             if (!AreEqual(GetHeaderReference().FirstFreeLink, linkIndex)) // May be this check
                    ↪  is not needed
454             {
455                 ref var link = ref GetLinkReference(linkIndex);
456                 return AreEqual(link.SizeAsSource, default) && !AreEqual(link.Source, default);
457             }
458             else
459             {
460                 return true;
461             }
462         }
463 
464         [MethodImpl(MethodImplOptions.AggressiveInlining)]
465         protected virtual TLink GetOne() => _one;
466 
467         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
468         protected virtual TLink GetZero() => default;
469
470         [MethodImpl(MethodImplOptions.AggressiveInlining)]
471         protected virtual bool AreEqual(TLink first, TLink second) =>
            ↪ _equalityComparer.Equals(first, second);
472
473         [MethodImpl(MethodImplOptions.AggressiveInlining)]
474         protected virtual bool LessThan(TLink first, TLink second) => _comparer.Compare(first,
            ↪ second) < 0;
475
476         [MethodImpl(MethodImplOptions.AggressiveInlining)]
477         protected virtual bool LessOrEqualThan(TLink first, TLink second) =>
            ↪ _comparer.Compare(first, second) <= 0;
478
479         [MethodImpl(MethodImplOptions.AggressiveInlining)]
480         protected virtual bool GreaterThan(TLink first, TLink second) =>
            ↪ _comparer.Compare(first, second) > 0;
481
482         [MethodImpl(MethodImplOptions.AggressiveInlining)]
483         protected virtual bool GreaterOrEqualThan(TLink first, TLink second) =>
            ↪ _comparer.Compare(first, second) >= 0;
484
485         [MethodImpl(MethodImplOptions.AggressiveInlining)]
486         protected virtual long ConvertToInt64(TLink value) =>
            ↪ _addressToInt64Converter.Convert(value);
487
488         [MethodImpl(MethodImplOptions.AggressiveInlining)]
489         protected virtual TLink ConvertToAddress(long value) =>
            ↪ _int64ToAddressConverter.Convert(value);
490
491         [MethodImpl(MethodImplOptions.AggressiveInlining)]
492         protected virtual TLink Add(TLink first, TLink second) => Arithmetic<TLink>.Add(first,
            ↪ second);
493
494         [MethodImpl(MethodImplOptions.AggressiveInlining)]
495         protected virtual TLink Subtract(TLink first, TLink second) =>
            ↪ Arithmetic<TLink>.Subtract(first, second);
496
497         [MethodImpl(MethodImplOptions.AggressiveInlining)]
498         protected virtual TLink Increment(TLink link) => Arithmetic<TLink>.Increment(link);
499
500         [MethodImpl(MethodImplOptions.AggressiveInlining)]
501         protected virtual TLink Decrement(TLink link) => Arithmetic<TLink>.Decrement(link);
502
503         [MethodImpl(MethodImplOptions.AggressiveInlining)]
504         protected virtual IList<TLink> GetEmptyList() => Array.Empty<TLink>();
505
506         #region Disposable
507
508         protected override bool AllowMultipleDisposeCalls => true;
509
510         protected override void Dispose(bool manual, bool wasDisposed)
511         {
512             if (!wasDisposed)
513             {
514                 ResetPointers();
515                 _memory.DisposeIfPossible();
516             }
517         }
518
519         #endregion
520     }
521 }
```

1.40 ./Platform.Data.Doublets/ResizableDirectMemory/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.ResizableDirectMemory.Generic
9  {
10     public unsafe class UnusedLinksListMethods<TLink> : CircularDoublyLinkedListMethods<TLink>,
       ↪ ILinksListMethods<TLink>
11     {
12         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
           ↪ UncheckedConverter<TLink, long>.Default;
```

```

        private readonly byte* _links;
        private readonly byte* _header;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnusedLinksListMethods(byte* links, byte* header)
        {
            _links = links;
            _header = header;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref LinksHeader<TLink> GetHeaderReference() => ref
        ↪  AsRef<LinksHeader<TLink>>(_header);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual ref RawLink<TLink> GetLinkReference(TLink link) => ref
        ↪  AsRef<RawLink<TLink>>(_links + (RawLink<TLink>.SizeInBytes *
        ↪  _addressToInt64Converter.Convert(link)));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetFirst() => GetHeaderReference().FirstFreeLink;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetLast() => GetHeaderReference().LastFreeLink;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetPrevious(TLink element) => GetLinkReference(element).Source;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNext(TLink element) => GetLinkReference(element).Target;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetSize() => GetHeaderReference().FreeLinks;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetFirst(TLink element) => GetHeaderReference().FirstFreeLink =
        ↪  element;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLast(TLink element) => GetHeaderReference().LastFreeLink =
        ↪  element;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetPrevious(TLink element, TLink previous) =>
        ↪  GetLinkReference(element).Source = previous;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetNext(TLink element, TLink next) =>
        ↪  GetLinkReference(element).Target = next;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSize(TLink size) => GetHeaderReference().FreeLinks = size;
    }
}
```

1.41 ./Platform.Data.Doublets/ResizableDirectMemory/ILinksListMethods.cs

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    public interface ILinksListMethods<TLink>
    {
        void Detach(TLink freeLink);
        void AttachAsFirst(TLink link);
    }
}
```

1.42 ./Platform.Data.Doublets/ResizableDirectMemory/ILinksTreeMethods.cs

```
using System;
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    public interface ILinksTreeMethods<TLink>
    {
        TLink CountUsages(TLink link);
```

```
        TLink Search(TLink source, TLink target);
        TLink EachUsage(TLink source, Func<IList<TLink>, TLink> handler);
        void Detach(ref TLink firstAsSource, TLink linkIndex);
        void Attach(ref TLink firstAsSource, TLink linkIndex);
    }
}
```

1.43 ./Platform.Data.Doublets/ResizableDirectMemory/LinksHeader.cs

```
using System;
using System.Collections.Generic;
using Platform.Unsafe;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    public struct LinksHeader<TLink> : IEquatable<LinksHeader<TLink>>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        public static readonly long SizeInBytes = Structure<LinksHeader<TLink>>.Size;

        public TLink AllocatedLinks;
        public TLink ReservedLinks;
        public TLink FreeLinks;
        public TLink FirstFreeLink;
        public TLink FirstAsSource;
        public TLink FirstAsTarget;
        public TLink LastFreeLink;
        public TLink Reserved8;

        public override bool Equals(object obj) => obj is LinksHeader<TLink> linksHeader ?
        ↪  Equals(linksHeader) : false;

        public bool Equals(LinksHeader<TLink> other)
            => _equalityComparer.Equals(AllocatedLinks, other.AllocatedLinks)
            && _equalityComparer.Equals(ReservedLinks, other.ReservedLinks)
            && _equalityComparer.Equals(FreeLinks, other.FreeLinks)
            && _equalityComparer.Equals(FirstFreeLink, other.FirstFreeLink)
            && _equalityComparer.Equals(FirstAsSource, other.FirstAsSource)
            && _equalityComparer.Equals(FirstAsTarget, other.FirstAsTarget)
            && _equalityComparer.Equals(LastFreeLink, other.LastFreeLink)
            && _equalityComparer.Equals(Reserved8, other.Reserved8);

        public override int GetHashCode() => (AllocatedLinks, ReservedLinks, FreeLinks,
        ↪  FirstFreeLink, FirstAsSource, FirstAsTarget, LastFreeLink, Reserved8).GetHashCode();

        public static bool operator ==(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
        ↪  left.Equals(right);

        public static bool operator !=(LinksHeader<TLink> left, LinksHeader<TLink> right) =>
        ↪  !(left == right);
    }
}
```

1.44 ./Platform.Data.Doublets/ResizableDirectMemory/RawLink.cs

```
using Platform.Unsafe;
using System;
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.ResizableDirectMemory
{
    public struct RawLink<TLink> : IEquatable<RawLink<TLink>>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        public static readonly long SizeInBytes = Structure<RawLink<TLink>>.Size;

        public TLink Source;
        public TLink Target;
        public TLink LeftAsSource;
        public TLink RightAsSource;
        public TLink SizeAsSource;
        public TLink LeftAsTarget;
        public TLink RightAsTarget;
        public TLink SizeAsTarget;

```

```
24          public override bool Equals(object obj) => obj is RawLink<TLink> link ? Equals(link) :
            ↪ false;
25
26          public bool Equals(RawLink<TLink> other)
27              => _equalityComparer.Equals(Source, other.Source)
28              && _equalityComparer.Equals(Target, other.Target)
29              && _equalityComparer.Equals(LeftAsSource, other.LeftAsSource)
30              && _equalityComparer.Equals(RightAsSource, other.RightAsSource)
31              && _equalityComparer.Equals(SizeAsSource, other.SizeAsSource)
32              && _equalityComparer.Equals(LeftAsTarget, other.LeftAsTarget)
33              && _equalityComparer.Equals(RightAsTarget, other.RightAsTarget)
34              && _equalityComparer.Equals(SizeAsTarget, other.SizeAsTarget);
35
36          public override int GetHashCode() => (Source, Target, LeftAsSource, RightAsSource,
            ↪ SizeAsSource, LeftAsTarget, RightAsTarget, SizeAsTarget).GetHashCode();
37
38          public static bool operator ==(RawLink<TLink> left, RawLink<TLink> right) =>
            ↪ left.Equals(right);
39
40          public static bool operator !=(RawLink<TLink> left, RawLink<TLink> right) => !(left ==
            ↪ right);
41      }
42  }
```

1.45 ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksAvlBalancedTreeMethodsBase.cs

```
1   using System.Runtime.CompilerServices;
2   using Platform.Data.Doublets.ResizableDirectMemory.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.ResizableDirectMemory.Specific
8   {
9       public unsafe abstract class UInt64LinksAvlBalancedTreeMethodsBase :
            ↪ LinksAvlBalancedTreeMethodsBase<ulong>
10      {
11          protected new readonly RawLink<ulong>* Links;
12          protected new readonly LinksHeader<ulong>* Header;
13
14          protected UInt64LinksAvlBalancedTreeMethodsBase(LinksConstants<ulong> constants,
            ↪ RawLink<ulong>* links, LinksHeader<ulong>* header)
15              : base(constants, (byte*)links, (byte*)header)
16          {
17              Links = links;
18              Header = header;
19          }
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          protected override ulong GetZero() => 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) => unchecked((value & 4294967264UL)
            ↪  >> 5);
84
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          protected override void SetSizeValue(ref ulong storedValue, ulong size) => storedValue =
            ↪  unchecked(storedValue & 31UL | (size & 134217727UL) << 5);
87
88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
89          protected override bool GetLeftIsChildValue(ulong value) => unchecked((value & 16UL) >>
            ↪  4 == 1UL);
90
91          [MethodImpl(MethodImplOptions.AggressiveInlining)]
92          protected override void SetLeftIsChildValue(ref ulong storedValue, bool value) =>
            ↪  storedValue = unchecked(storedValue & 4294967279UL | (As<bool, byte>(ref value) &
            ↪  1UL) << 4);
93
94          [MethodImpl(MethodImplOptions.AggressiveInlining)]
95          protected override bool GetRightIsChildValue(ulong value) => unchecked((value & 8UL) >>
            ↪  3 == 1UL);
96
97          [MethodImpl(MethodImplOptions.AggressiveInlining)]
98          protected override void SetRightIsChildValue(ref ulong storedValue, bool value) =>
            ↪  storedValue = unchecked(storedValue & 4294967287UL | (As<bool, byte>(ref value) &
            ↪  1UL) << 3);
99
100         [MethodImpl(MethodImplOptions.AggressiveInlining)]
101         protected override sbyte GetBalanceValue(ulong value) => unchecked((sbyte)(value & 7UL |
            ↪  0xF8UL * ((value & 4UL) >> 2))); // if negative, then continue ones to the end of
            ↪  sbyte
102
103         [MethodImpl(MethodImplOptions.AggressiveInlining)]
104         protected override void SetBalanceValue(ref ulong storedValue, sbyte value) =>
            ↪  storedValue = unchecked(storedValue & 4294967288UL | (ulong)((byte)value >> 5 & 4 |
            ↪  value & 3) & 7UL);
105
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
108
109         [MethodImpl(MethodImplOptions.AggressiveInlining)]
110         protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
111     }
112 }
```

```
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.ResizableDirectMemory.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
{
    public unsafe abstract class UInt64LinksSizeBalancedTreeMethodsBase :
    ↪ LinksSizeBalancedTreeMethodsBase<ulong>
    {
        protected new readonly RawLink<ulong>* Links;
        protected new readonly LinksHeader<ulong>* Header;

        protected UInt64LinksSizeBalancedTreeMethodsBase(LinksConstants<ulong> constants,
        ↪ RawLink<ulong>* links, LinksHeader<ulong>* header)
            : base(constants, (byte*)links, (byte*)header)
        {
            Links = links;
            Header = header;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetZero() => 0UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool EqualToZero(ulong value) => value == 0UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool AreEqual(ulong first, ulong second) => first == second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThanZero(ulong value) => value > 0UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterThan(ulong first, ulong second) => first > second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0 is
        ↪ always true for ulong

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThanZero(ulong value) => value == 0UL; // value is
        ↪ always >= 0 for ulong

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThanZero(ulong value) => false; // value < 0 is always false
        ↪ for ulong

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool LessThan(ulong first, ulong second) => first < second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Increment(ulong value) => ++value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Decrement(ulong value) => --value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Add(ulong first, ulong second) => first + second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong Subtract(ulong first, ulong second) => first - second;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
        {
            ref var firstLink = ref Links[first];
            ref var secondLink = ref Links[second];
            return FirstIsToTheLeftOfSecond(firstLink.Source, firstLink.Target,
            ↪ secondLink.Source, secondLink.Target);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
74          protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
75          {
76              ref var firstLink = ref Links[first];
77              ref var secondLink = ref Links[second];
78              return FirstIsToTheRightOfSecond(firstLink.Source, firstLink.Target,
                ↪  secondLink.Source, secondLink.Target);
79          }
80
81          [MethodImpl(MethodImplOptions.AggressiveInlining)]
82          protected override ref LinksHeader<ulong> GetHeaderReference() => ref *Header;
83
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          protected override ref RawLink<ulong> GetLinkReference(ulong link) => ref Links[link];
86      }
87  }
```

1.47 ./Platform.Data.Doublets/ResizableDirectMemory/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.ResizableDirectMemory.Specific
6   {
7       public unsafe class UInt64LinksSourcesAvlBalancedTreeMethods :
        ↪  UInt64LinksAvlBalancedTreeMethodsBase
8       {
9           public UInt64LinksSourcesAvlBalancedTreeMethods(LinksConstants<ulong> constants,
            ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
            ↪  { }
10
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected override ref ulong GetLeftReference(ulong node) => ref
            ↪  Links[node].LeftAsSource;
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          protected override ref ulong GetRightReference(ulong node) => ref
            ↪  Links[node].RightAsSource;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource =
            ↪  left;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override void SetRight(ulong node, ulong right) => Links[node].RightAsSource =
            ↪  right;
28
29          [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
            ↪  Links[node].SizeAsSource, size);
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected override bool GetLeftIsChild(ulong node) =>
            ↪  GetLeftIsChildValue(Links[node].SizeAsSource);
37
38          //[MethodImpl(MethodImplOptions.AggressiveInlining)]
39          //protected override bool GetLeftIsChild(ulong node) => IsChild(node, GetLeft(node));
40
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          protected override void SetLeftIsChild(ulong node, bool value) =>
            ↪  SetLeftIsChildValue(ref Links[node].SizeAsSource, value);
43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          protected override bool GetRightIsChild(ulong node) =>
            ↪  GetRightIsChildValue(Links[node].SizeAsSource);
46
47          //[MethodImpl(MethodImplOptions.AggressiveInlining)]
48          //protected override bool GetRightIsChild(ulong node) => IsChild(node, GetRight(node));
49
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
51          protected override void SetRightIsChild(ulong node, bool value) =>
            ↪ SetRightIsChildValue(ref Links[node].SizeAsSource, value);
52
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          protected override sbyte GetBalance(ulong node) =>
            ↪ GetBalanceValue(Links[node].SizeAsSource);
55
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
            ↪ Links[node].SizeAsSource, value);
58
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          protected override ulong GetTreeRoot() => Header->FirstAsSource;
61
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
64
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          protected override bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
            ↪ ulong secondSource, ulong secondTarget)
67              => firstSource < secondSource || (firstSource == secondSource && firstTarget <
                ↪ secondTarget);
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
            ↪ ulong secondSource, ulong secondTarget)
71              => firstSource > secondSource || (firstSource == secondSource && firstTarget >
                ↪ secondTarget);
72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override void ClearNode(ulong node)
75          {
76              ref var link = ref Links[node];
77              link.LeftAsSource = 0UL;
78              link.RightAsSource = 0UL;
79              link.SizeAsSource = 0UL;
80          }
81      }
82  }
```

1.48 ./Platform.Data.Doublets/ResizableDirectMemory/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.ResizableDirectMemory.Specific
6   {
7       public unsafe class UInt64LinksSourcesSizeBalancedTreeMethods :
        ↪ UInt64LinksSizeBalancedTreeMethodsBase
8       {
9           public UInt64LinksSourcesSizeBalancedTreeMethods(LinksConstants<ulong> constants,
            ↪ RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
            ↪ { }
10
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected override ref ulong GetLeftReference(ulong node) => ref
            ↪ Links[node].LeftAsSource;
13
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->FirstAsSource;
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 = 0UL;
54              link.RightAsSource = 0UL;
55              link.SizeAsSource = 0UL;
56          }
57      }
58  }
```

1.49 ./Platform.Data.Doublets/ResizableDirectMemory/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.ResizableDirectMemory.Specific
6   {
7       public unsafe class UInt64LinksTargetsAvlBalancedTreeMethods :
        ↪  UInt64LinksAvlBalancedTreeMethodsBase
8       {
9           public UInt64LinksTargetsAvlBalancedTreeMethods(LinksConstants<ulong> constants,
            ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
            ↪  { }
10
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          protected override ref ulong GetLeftReference(ulong node) => ref
            ↪  Links[node].LeftAsTarget;
13
14          [MethodImpl(MethodImplOptions.AggressiveInlining)]
15          protected override ref ulong GetRightReference(ulong node) => ref
            ↪  Links[node].RightAsTarget;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget =
            ↪  left;
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          protected override void SetRight(ulong node, ulong right) => Links[node].RightAsTarget =
            ↪  right;
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          protected override ulong GetSize(ulong node) => GetSizeValue(Links[node].SizeAsTarget);
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          protected override void SetSize(ulong node, ulong size) => SetSizeValue(ref
            ↪  Links[node].SizeAsTarget, size);
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          protected override bool GetLeftIsChild(ulong node) =>
            ↪  GetLeftIsChildValue(Links[node].SizeAsTarget);
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetLeftIsChild(ulong node, bool value) =>
        ↪  SetLeftIsChildValue(ref Links[node].SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool GetRightIsChild(ulong node) =>
        ↪  GetRightIsChildValue(Links[node].SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetRightIsChild(ulong node, bool value) =>
        ↪  SetRightIsChildValue(ref Links[node].SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override sbyte GetBalance(ulong node) =>
        ↪  GetBalanceValue(Links[node].SizeAsTarget);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetBalance(ulong node, sbyte value) => SetBalanceValue(ref
        ↪  Links[node].SizeAsTarget, value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ulong GetTreeRoot() => Header->FirstAsTarget;

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
        ↪  ulong secondSource, ulong secondTarget)
            => firstTarget > secondTarget || (firstTarget == secondTarget && firstSource >
            ↪  secondSource);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void ClearNode(ulong node)
        {
            ref var link = ref Links[node];
            link.LeftAsTarget = 0UL;
            link.RightAsTarget = 0UL;
            link.SizeAsTarget = 0UL;
        }
    }
}
```

1.50 ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64LinksTargetsSizeBalancedTreeMethods.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
{
    public unsafe class UInt64LinksTargetsSizeBalancedTreeMethods :
    ↪  UInt64LinksSizeBalancedTreeMethodsBase
    {
        public UInt64LinksTargetsSizeBalancedTreeMethods(LinksConstants<ulong> constants,
        ↪  RawLink<ulong>* links, LinksHeader<ulong>* header) : base(constants, links, header)
        ↪  { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetLeftReference(ulong node) => ref
        ↪  Links[node].LeftAsTarget;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override ref ulong GetRightReference(ulong node) => ref
        ↪  Links[node].RightAsTarget;

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

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

```

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

1.51 ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64ResizableDirectMemoryLinks.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Memory;
5   using Platform.Data.Doublets.ResizableDirectMemory.Generic;
6   using Platform.Singletons;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets.ResizableDirectMemory.Specific
11  {
12      public unsafe class UInt64ResizableDirectMemoryLinks : ResizableDirectMemoryLinksBase<ulong>
13      {
14          private readonly Func<ILinksTreeMethods<ulong>> _createSourceTreeMethods;
15          private readonly Func<ILinksTreeMethods<ulong>> _createTargetTreeMethods;
16          private LinksHeader<ulong>* _header;
17          private RawLink<ulong>* _links;
18
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          public UInt64ResizableDirectMemoryLinks(string address) : this(address,
            ↪  DefaultLinksSizeStep) { }
21
22          /// <summary>
23          /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
            ↪  минимальным шагом расширения базы данных.
24          /// </summary>
25          /// <param name="address">Полный пусть к файлу базы данных.</param>
26          /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
            ↪  байтах.</param>
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
            ↪  this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
            ↪  memoryReservationStep) { }
29
```

```
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
            ↪  DefaultLinksSizeStep) { }
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep) : this(memory, memoryReservationStep,
            ↪  Default<LinksConstants<ulong>>.Instance, true) { }
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
            ↪  memoryReservationStep, LinksConstants<ulong> constants, bool useAvlBasedIndex) :
            ↪  base(memory, memoryReservationStep, constants)
38          {
39              if (useAvlBasedIndex)
40              {
41                  _createSourceTreeMethods = () => new
                    ↪  UInt64LinksSourcesAvlBalancedTreeMethods(Constants, _links, _header);
42                  _createTargetTreeMethods = () => new
                    ↪  UInt64LinksTargetsAvlBalancedTreeMethods(Constants, _links, _header);
43              }
44              else
45              {
46                  _createSourceTreeMethods = () => new
                    ↪  UInt64LinksSourcesSizeBalancedTreeMethods(Constants, _links, _header);
47                  _createTargetTreeMethods = () => new
                    ↪  UInt64LinksTargetsSizeBalancedTreeMethods(Constants, _links, _header);
48              }
49              Init(memory, memoryReservationStep);
50          }
51
52          [MethodImpl(MethodImplOptions.AggressiveInlining)]
53          protected override void SetPointers(IResizableDirectMemory memory)
54          {
55              _header = (LinksHeader<ulong>*)memory.Pointer;
56              _links = (RawLink<ulong>*)memory.Pointer;
57              SourcesTreeMethods = _createSourceTreeMethods();
58              TargetsTreeMethods = _createTargetTreeMethods();
59              UnusedLinksListMethods = new UInt64UnusedLinksListMethods(_links, _header);
60          }
61
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          protected override void ResetPointers()
64          {
65              base.ResetPointers();
66              _links = null;
67              _header = null;
68          }
69
70          [MethodImpl(MethodImplOptions.AggressiveInlining)]
71          protected override ref LinksHeader<ulong> GetHeaderReference() => ref *_header;
72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          protected override ref RawLink<ulong> GetLinkReference(ulong linkIndex) => ref
            ↪  _links[linkIndex];
75
76          [MethodImpl(MethodImplOptions.AggressiveInlining)]
77          protected override bool AreEqual(ulong first, ulong second) => first == second;
78
79          [MethodImpl(MethodImplOptions.AggressiveInlining)]
80          protected override bool LessThan(ulong first, ulong second) => first < second;
81
82          [MethodImpl(MethodImplOptions.AggressiveInlining)]
83          protected override bool LessOrEqualThan(ulong first, ulong second) => first <= second;
84
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          protected override bool GreaterThan(ulong first, ulong second) => first > second;
87
88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
89          protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >= second;
90
91          [MethodImpl(MethodImplOptions.AggressiveInlining)]
92          protected override ulong GetZero() => 0UL;
93
94          [MethodImpl(MethodImplOptions.AggressiveInlining)]
95          protected override ulong GetOne() => 1UL;
96
97          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
98          protected override long ConvertToInt64(ulong value) => (long)value;
99
100         [MethodImpl(MethodImplOptions.AggressiveInlining)]
101         protected override ulong ConvertToAddress(long value) => (ulong)value;
102
103         [MethodImpl(MethodImplOptions.AggressiveInlining)]
104         protected override ulong Add(ulong first, ulong second) => first + second;
105
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         protected override ulong Subtract(ulong first, ulong second) => first - second;
108
109         [MethodImpl(MethodImplOptions.AggressiveInlining)]
110         protected override ulong Increment(ulong link) => ++link;
111
112         [MethodImpl(MethodImplOptions.AggressiveInlining)]
113         protected override ulong Decrement(ulong link) => --link;
114     }
115 }
```

1.52 ./Platform.Data.Doublets/ResizableDirectMemory/Specific/UInt64UnusedLinksListMethods.cs

```
1  using System.Runtime.CompilerServices;
2  using Platform.Data.Doublets.ResizableDirectMemory.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.ResizableDirectMemory.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 }
```

1.53 ./Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs

```
1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Converters
6  {
7      public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
8      {
9          public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }
10
11         public override TLink Convert(IList<TLink> sequence)
12         {
13             var length = sequence.Count;
14             if (length < 1)
15             {
16                 return default;
17             }
18             if (length == 1)
19             {
20                 return sequence[0];
21             }
22             // Make copy of next layer
23             if (length > 2)
24             {
25                 // TODO: Try to use stackalloc (which at the moment is not working with
                   ↪  generics) but will be possible with Sigil
26                 var halvedSequence = new TLink[(length / 2) + (length % 2)];
27                 HalveSequence(halvedSequence, sequence, length);
28                 sequence = halvedSequence;
29                 length = halvedSequence.Length;
30             }
```

```
            // Keep creating layer after layer
            while (length > 2)
            {
                HalveSequence(sequence, sequence, length);
                length = (length / 2) + (length % 2);
            }
            return Links.GetOrCreate(sequence[0], sequence[1]);
        }

        private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
        {
            var loopedLength = length - (length % 2);
            for (var i = 0; i < loopedLength; i += 2)
            {
                destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
            }
            if (length > loopedLength)
            {
                destination[length / 2] = source[length - 1];
            }
        }
    }
}
```

1.54 ./Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections;
using Platform.Converters;
using Platform.Singletons;
using Platform.Numbers;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Converters
{
    /// <remarks>
    /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от Links на этапе сжатия.
    ///     А именно будет создаваться временный список пар необходимых для выполнения сжатия, в таком случае тип значения элемента массива может быть любым, как char так и ulong.
    ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих пар, а так же разом выполнить замену.
    /// </remarks>
    public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
    {
        private static readonly LinksConstants<TLink> _constants = Default<LinksConstants<TLink>>.Instance;
        private static readonly EqualityComparer<TLink> _equalityComparer = EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        private readonly IConverter<IList<TLink>, TLink> _baseConverter;
        private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
        private readonly TLink _minFrequencyToCompress;
        private readonly bool _doInitialFrequenciesIncrement;
        private Doublet<TLink> _maxDoublet;
        private LinkFrequency<TLink> _maxDoubletData;

        private struct HalfDoublet
        {
            public TLink Element;
            public LinkFrequency<TLink> DoubletData;

            public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
            {
                Element = element;
                DoubletData = doubletData;
            }

            public override string ToString() => $"{Element}: ({DoubletData})";
        }

        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink> baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
            : this(links, baseConverter, doubletFrequenciesCache, _one, true)
```

```
51          {
52          }
53
54          public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
            ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
            ↪  doInitialFrequenciesIncrement)
55              : this(links, baseConverter, doubletFrequenciesCache, _one,
                ↪  doInitialFrequenciesIncrement)
56          {
57          }
58
59          public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
            ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
            ↪  minFrequencyToCompress, bool doInitialFrequenciesIncrement)
60              : base(links)
61          {
62              _baseConverter = baseConverter;
63              _doubletFrequenciesCache = doubletFrequenciesCache;
64              if (_comparer.Compare(minFrequencyToCompress, _one) < 0)
65              {
66                  minFrequencyToCompress = _one;
67              }
68              _minFrequencyToCompress = minFrequencyToCompress;
69              _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
70              ResetMaxDoublet();
71          }
72
73          public override TLink Convert(IList<TLink> source) =>
            ↪  _baseConverter.Convert(Compress(source));
74
75          /// <remarks>
76          /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding .
77          /// Faster version (doublets' frequencies dictionary is not recreated).
78          /// </remarks>
79          private IList<TLink> Compress(IList<TLink> sequence)
80          {
81              if (sequence.IsNullOrEmpty())
82              {
83                  return null;
84              }
85              if (sequence.Count == 1)
86              {
87                  return sequence;
88              }
89              if (sequence.Count == 2)
90              {
91                  return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
92              }
93              // TODO: arraypool with min size (to improve cache locality) or stackallow with Sigil
94              var copy = new HalfDoublet[sequence.Count];
95              Doublet<TLink> doublet = default;
96              for (var i = 1; i < sequence.Count; i++)
97              {
98                  doublet.Source = sequence[i - 1];
99                  doublet.Target = sequence[i];
100                 LinkFrequency<TLink> data;
101                 if (_doInitialFrequenciesIncrement)
102                 {
103                     data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
104                 }
105                 else
106                 {
107                     data = _doubletFrequenciesCache.GetFrequency(ref doublet);
108                     if (data == null)
109                     {
110                         throw new NotSupportedException("If you ask not to increment
                            ↪  frequencies, it is expected that all frequencies for the sequence
                            ↪  are prepared.");
111                     }
112                 }
113                 copy[i - 1].Element = sequence[i - 1];
114                 copy[i - 1].DoubletData = data;
115                 UpdateMaxDoublet(ref doublet, data);
116             }
117             copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
118             copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
119             if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
120             {
```

```
                var newLength = ReplaceDoublets(copy);
                sequence = new TLink[newLength];
                for (int i = 0; i < newLength; i++)
                {
                    sequence[i] = copy[i].Element;
                }
            }
            return sequence;
        }

        /// <remarks>
        /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding
        /// </remarks>
        private int ReplaceDoublets(HalfDoublet[] copy)
        {
            var oldLength = copy.Length;
            var newLength = copy.Length;
            while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
            {
                var maxDoubletSource = _maxDoublet.Source;
                var maxDoubletTarget = _maxDoublet.Target;
                if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
                {
                    _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
                }
                var maxDoubletReplacementLink = _maxDoubletData.Link;
                oldLength--;
                var oldLengthMinusTwo = oldLength - 1;
                // Substitute all usages
                int w = 0, r = 0; // (r == read, w == write)
                for (; r < oldLength; r++)
                {
                    if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
                    ↪  _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
                    {
                        if (r > 0)
                        {
                            var previous = copy[w - 1].Element;
                            copy[w - 1].DoubletData.DecrementFrequency();
                            copy[w - 1].DoubletData =
                            ↪  _doubletFrequenciesCache.IncrementFrequency(previous,
                            ↪  maxDoubletReplacementLink);
                        }
                        if (r < oldLengthMinusTwo)
                        {
                            var next = copy[r + 2].Element;
                            copy[r + 1].DoubletData.DecrementFrequency();
                            copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(ma
                            ↪  xDoubletReplacementLink,
                            ↪  next);
                        }
                        copy[w++].Element = maxDoubletReplacementLink;
                        r++;
                        newLength--;
                    }
                    else
                    {
                        copy[w++] = copy[r];
                    }
                }
                if (w < newLength)
                {
                    copy[w] = copy[r];
                }
                oldLength = newLength;
                ResetMaxDoublet();
                UpdateMaxDoublet(copy, newLength);
            }
            return newLength;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void ResetMaxDoublet()
        {
            _maxDoublet = new Doublet<TLink>();
            _maxDoubletData = new LinkFrequency<TLink>();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
195         private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
196         {
197             Doublet<TLink> doublet = default;
198             for (var i = 1; i < length; i++)
199             {
200                 doublet.Source = copy[i - 1].Element;
201                 doublet.Target = copy[i].Element;
202                 UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
203             }
204         }
205 
206         [MethodImpl(MethodImplOptions.AggressiveInlining)]
207         private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
208         {
209             var frequency = data.Frequency;
210             var maxFrequency = _maxDoubletData.Frequency;
211             //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
                ↪  (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
                ↪  compression string data (and gives collisions quickly) */ _maxDoublet.Source +
                ↪  _maxDoublet.Target)))
212             if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
213                (_comparer.Compare(maxFrequency, frequency) < 0 ||
                    ↪  (_equalityComparer.Equals(maxFrequency, frequency) &&
                    ↪  _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
                    ↪  Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
                    ↪  better stability and better compression on sequent data and even on rundom
                    ↪  numbers data (but gives collisions anyway) */
214             {
215                 _maxDoublet = doublet;
216                 _maxDoubletData = data;
217             }
218         }
219     }
220 }
```

1.55 ./Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs

```
1  using System.Collections.Generic;
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.Converters
7  {
8      public abstract class LinksListToSequenceConverterBase<TLink> : IConverter<IList<TLink>,
       ↪  TLink>
9      {
10         protected readonly ILinks<TLink> Links;
11 
12         protected LinksListToSequenceConverterBase(ILinks<TLink> links) => Links = links;
13 
14         public abstract TLink Convert(IList<TLink> source);
15     }
16 }
```

1.56 ./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs

```
1  using System.Collections.Generic;
2  using System.Linq;
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 OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
12         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
13 
14         private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
15 
16         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
           ↪  sequenceToItsLocalElementLevelsConverter) : base(links)
17             => _sequenceToItsLocalElementLevelsConverter =
                ↪  sequenceToItsLocalElementLevelsConverter;
18 
19         public override TLink Convert(IList<TLink> sequence)
20         {
21             var length = sequence.Count;
```

```
            if (length == 1)
            {
                return sequence[0];
            }
            var links = Links;
            if (length == 2)
            {
                return links.GetOrCreate(sequence[0], sequence[1]);
            }
            sequence = sequence.ToArray();
            var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
            while (length > 2)
            {
                var levelRepeat = 1;
                var currentLevel = levels[0];
                var previousLevel = levels[0];
                var skipOnce = false;
                var w = 0;
                for (var i = 1; i < length; i++)
                {
                    if (_equalityComparer.Equals(currentLevel, levels[i]))
                    {
                        levelRepeat++;
                        skipOnce = false;
                        if (levelRepeat == 2)
                        {
                            sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
                            var newLevel = i >= length - 1 ?
                                GetPreviousLowerThanCurrentOrCurrent(previousLevel,
                                ↪ currentLevel) :
                                i < 2 ?
                                GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
                                GetGreatestNeigbourLowerThanCurrentOrCurrent(previousLevel,
                                ↪ currentLevel, levels[i + 1]);
                            levels[w] = newLevel;
                            previousLevel = currentLevel;
                            w++;
                            levelRepeat = 0;
                            skipOnce = true;
                        }
                        else if (i == length - 1)
                        {
                            sequence[w] = sequence[i];
                            levels[w] = levels[i];
                            w++;
                        }
                    }
                    else
                    {
                        currentLevel = levels[i];
                        levelRepeat = 1;
                        if (skipOnce)
                        {
                            skipOnce = false;
                        }
                        else
                        {
                            sequence[w] = sequence[i - 1];
                            levels[w] = levels[i - 1];
                            previousLevel = levels[w];
                            w++;
                        }
                        if (i == length - 1)
                        {
                            sequence[w] = sequence[i];
                            levels[w] = levels[i];
                            w++;
                        }
                    }
                }
                length = w;
            }
            return links.GetOrCreate(sequence[0], sequence[1]);
        }

        private static TLink GetGreatestNeigbourLowerThanCurrentOrCurrent(TLink previous, TLink
        ↪ current, TLink next)
        {
            return _comparer.Compare(previous, next) > 0
```

```
                ? _comparer.Compare(previous, current) < 0 ? previous : current
                : _comparer.Compare(next, current) < 0 ? next : current;
        }

        private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
        ↪  _comparer.Compare(next, current) < 0 ? next : current;

        private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
        ↪  => _comparer.Compare(previous, current) < 0 ? previous : current;
    }
}
```

1.57 ./Platform.Data.Doublets/Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs

```
using System.Collections.Generic;
using Platform.Converters;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Converters
{
    public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<IList<TLink>>
    {
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;

        public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
        ↪  IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
        ↪  => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;

        public IList<TLink> Convert(IList<TLink> sequence)
        {
            var levels = new TLink[sequence.Count];
            levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
            for (var i = 1; i < sequence.Count - 1; i++)
            {
                var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
                var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
                levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
            }
            levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
            ↪  sequence[sequence.Count - 1]);
            return levels;
        }

        public TLink GetFrequencyNumber(TLink source, TLink target) =>
        ↪  _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
    }
}
```

1.58 ./Platform.Data.Doublets/Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs

```
using Platform.Interfaces;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.CriterionMatchers
{
    public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
    ↪  ICriterionMatcher<TLink>
    {
        public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
        public bool IsMatched(TLink argument) => Links.IsPartialPoint(argument);
    }
}
```

1.59 ./Platform.Data.Doublets/Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs

```
using System.Collections.Generic;
using Platform.Interfaces;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.CriterionMatchers
{
    public class MarkedSequenceCriterionMatcher<TLink> : ICriterionMatcher<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

```

```
        private readonly ILinks<TLink> _links;
        private readonly TLink _sequenceMarkerLink;

        public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
        {
            _links = links;
            _sequenceMarkerLink = sequenceMarkerLink;
        }

        public bool IsMatched(TLink sequenceCandidate)
            => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
            || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
            ↪  sequenceCandidate), _links.Constants.Null);
    }
}
```

1.60 ./Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs

```
using System.Collections.Generic;
using Platform.Collections.Stacks;
using Platform.Data.Doublets.Sequences.HeightProviders;
using Platform.Data.Sequences;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences
{
    public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
    ↪  ISequenceAppender<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly IStack<TLink> _stack;
        private readonly ISequenceHeightProvider<TLink> _heightProvider;

        public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
        ↪  ISequenceHeightProvider<TLink> heightProvider)
            : base(links)
        {
            _stack = stack;
            _heightProvider = heightProvider;
        }

        public TLink Append(TLink sequence, TLink appendant)
        {
            var cursor = sequence;
            while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
            {
                var source = Links.GetSource(cursor);
                var target = Links.GetTarget(cursor);
                if (_equalityComparer.Equals(_heightProvider.Get(source),
                ↪  _heightProvider.Get(target)))
                {
                    break;
                }
                else
                {
                    _stack.Push(source);
                    cursor = target;
                }
            }
            var left = cursor;
            var right = appendant;
            while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
            {
                right = Links.GetOrCreate(left, right);
                left = cursor;
            }
            return Links.GetOrCreate(left, right);
        }
    }
}
```

1.61 ./Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs

```
using System.Collections.Generic;
using System.Linq;
using Platform.Interfaces;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

```

```
namespace Platform.Data.Doublets.Sequences
{
    public class DuplicateSegmentsCounter<TLink> : ICounter<int>
    {
        private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
        ↪  _duplicateFragmentsProvider;
        public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
        ↪  IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
        ↪  duplicateFragmentsProvider;
        public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
    }
}
```

1.62 ./Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs

```
using System;
using System.Linq;
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Collections;
using Platform.Collections.Lists;
using Platform.Collections.Segments;
using Platform.Collections.Segments.Walkers;
using Platform.Singletons;
using Platform.Converters;
using Platform.Data.Doublets.Unicode;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences
{
    public class DuplicateSegmentsProvider<TLink> :
    ↪  DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
    ↪  IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
    {
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
        ↪  UncheckedConverter<TLink, long>.Default;
        private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
        ↪  UncheckedConverter<TLink, ulong>.Default;
        private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
        ↪  UncheckedConverter<ulong, TLink>.Default;

        private readonly ILinks<TLink> _links;
        private readonly ILinks<TLink> _sequences;
        private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
        private BitString _visited;

        private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
        ↪  IList<TLink>>>
        {
            private readonly IListEqualityComparer<TLink> _listComparer;
            public ItemEquilityComparer() => _listComparer =
            ↪  Default<IListEqualityComparer<TLink>>.Instance;
            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);
            public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
            ↪  (_listComparer.GetHashCode(pair.Key),
            ↪  _listComparer.GetHashCode(pair.Value)).GetHashCode();
        }

        private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
        {
            private readonly IListComparer<TLink> _listComparer;

            public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;

            public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
            ↪  KeyValuePair<IList<TLink>, IList<TLink>> right)
            {
                var intermediateResult = _listComparer.Compare(left.Key, right.Key);
                if (intermediateResult == 0)
                {
                    intermediateResult = _listComparer.Compare(left.Value, right.Value);
                }
                return intermediateResult;
            }
        }

        public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
```

```
54              : base(minimumStringSegmentLength: 2)
55          {
56              _links = links;
57              _sequences = sequences;
58          }
59
60          public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
61          {
62              _groups = new HashSet<KeyValuePair<IList<TLink>,
                ↪  IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
63              var count = _links.Count();
64              _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
65              _links.Each(link =>
66              {
67                  var linkIndex = _links.GetIndex(link);
68                  var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
69                  if (!_visited.Get(linkBitIndex))
70                  {
71                      var sequenceElements = new List<TLink>();
72                      var filler = new ListFiller<TLink, TLink>(sequenceElements,
                        ↪  _sequences.Constants.Break);
73                      _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
                        ↪  LinkAddress<TLink>(linkIndex));
74                      if (sequenceElements.Count > 2)
75                      {
76                          WalkAll(sequenceElements);
77                      }
78                  }
79                  return _links.Constants.Continue;
80              });
81              var resultList = _groups.ToList();
82              var comparer = Default<ItemComparer>.Instance;
83              resultList.Sort(comparer);
84  #if DEBUG
85              foreach (var item in resultList)
86              {
87                  PrintDuplicates(item);
88              }
89  #endif
90              return resultList;
91          }
92
93          protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
            ↪  length) => new Segment<TLink>(elements, offset, length);
94
95          protected override void OnDublicateFound(Segment<TLink> segment)
96          {
97              var duplicates = CollectDuplicatesForSegment(segment);
98              if (duplicates.Count > 1)
99              {
100                 _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
                    ↪  duplicates));
101             }
102         }
103
104         private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
105         {
106             var duplicates = new List<TLink>();
107             var readAsElement = new HashSet<TLink>();
108             var restrictions = segment.ShiftRight();
109             restrictions[0] = _sequences.Constants.Any;
110             _sequences.Each(sequence =>
111             {
112                 var sequenceIndex = sequence[_sequences.Constants.IndexPart];
113                 duplicates.Add(sequenceIndex);
114                 readAsElement.Add(sequenceIndex);
115                 return _sequences.Constants.Continue;
116             }, restrictions);
117             if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
118             {
119                 return new List<TLink>();
120             }
121             foreach (var duplicate in duplicates)
122             {
123                 var duplicateBitIndex = _addressToInt64Converter.Convert(duplicate);
124                 _visited.Set(duplicateBitIndex);
125             }
126             if (_sequences is Sequences sequencesExperiments)
```

```
            {
                var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((HashSet<ulong>)(object)readAsElement, (IList<ulong>)segment);
                foreach (var partiallyMatchedSequence in partiallyMatched)
                {
                    var sequenceIndex = _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
                    duplicates.Add(sequenceIndex);
                }
            }
            duplicates.Sort();
            return duplicates;
        }

        private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
        {
            if (!(_links is ILinks<ulong> ulongLinks))
            {
                return;
            }
            var duplicatesKey = duplicatesItem.Key;
            var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
            Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
            var duplicatesList = duplicatesItem.Value;
            for (int i = 0; i < duplicatesList.Count; i++)
            {
                var sequenceIndex = _addressToUInt64Converter.Convert(duplicatesList[i]);
                var formatedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x => Point<ulong>.IsPartialPoint(x), (sb, link) => _ = UnicodeMap.IsCharLink(link.Index) ? sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
                Console.WriteLine(formatedSequenceStructure);
                var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex, ulongLinks);
                Console.WriteLine(sequenceString);
            }
            Console.WriteLine();
        }
    }
}
```

1.63 ./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Numbers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    /// <remarks>
    /// Can be used to operate with many CompressingConverters (to keep global frequencies data between them).
    /// TODO: Extract interface to implement frequencies storage inside Links storage
    /// </remarks>
    public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer = EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
        private readonly ICounter<TLink, TLink> _frequencyCounter;

        public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
            : base(links)
        {
            _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096, DoubletComparer<TLink>.Default);
            _frequencyCounter = frequencyCounter;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
        {
            var doublet = new Doublet<TLink>(source, target);
            return GetFrequency(ref doublet);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
        {
            _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
            return data;
        }

        public void IncrementFrequencies(IList<TLink> sequence)
        {
            for (var i = 1; i < sequence.Count; i++)
            {
                IncrementFrequency(sequence[i - 1], sequence[i]);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
        {
            var doublet = new Doublet<TLink>(source, target);
            return IncrementFrequency(ref doublet);
        }

        public void PrintFrequencies(IList<TLink> sequence)
        {
            for (var i = 1; i < sequence.Count; i++)
            {
                PrintFrequency(sequence[i - 1], sequence[i]);
            }
        }

        public void PrintFrequency(TLink source, TLink target)
        {
            var number = GetFrequency(source, target).Frequency;
            Console.WriteLine("({0},{1}) - {2}", source, target, number);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
        {
            if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
            {
                data.IncrementFrequency();
            }
            else
            {
                var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
                data = new LinkFrequency<TLink>(_one, link);
                if (!_equalityComparer.Equals(link, default))
                {
                    data.Frequency = Arithmetic.Add(data.Frequency,
                    ↪  _frequencyCounter.Count(link));
                }
                _doubletsCache.Add(doublet, data);
            }
            return data;
        }

        public void ValidateFrequencies()
        {
            foreach (var entry in _doubletsCache)
            {
                var value = entry.Value;
                var linkIndex = value.Link;
                if (!_equalityComparer.Equals(linkIndex, default))
                {
                    var frequency = value.Frequency;
                    var count = _frequencyCounter.Count(linkIndex);
                    // TODO: Why `frequency` always greater than `count` by 1?
                    if (((_comparer.Compare(frequency, count) > 0) &&
                    ↪  (_comparer.Compare(Arithmetic.Subtract(frequency, count), _one) > 0))
                     || ((_comparer.Compare(count, frequency) > 0) &&
                     ↪  (_comparer.Compare(Arithmetic.Subtract(count, frequency), _one) > 0)))
```

```
                    {
                        throw new InvalidOperationException("Frequencies validation failed.");
                    }
                }
                //else
                //{
                //    if (value.Frequency > 0)
                //    {
                //        var frequency = value.Frequency;
                //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
                //        var count = _countLinkFrequency(linkIndex);

                //        if ((frequency > count && frequency - count > 1) || (count > frequency
                ↪  && count - frequency > 1))
                //            throw new Exception("Frequencies validation failed.");
                //    }
                //}
            }
        }
    }
}
```

1.64 ./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs

```
using System.Runtime.CompilerServices;
using Platform.Numbers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    public class LinkFrequency<TLink>
    {
        public TLink Frequency { get; set; }
        public TLink Link { get; set; }

        public LinkFrequency(TLink frequency, TLink link)
        {
            Frequency = frequency;
            Link = link;
        }

        public LinkFrequency() { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncrementFrequency() => Frequency = Arithmetic<TLink>.Increment(Frequency);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);

        public override string ToString() => $"F: {Frequency}, L: {Link}";
    }
}
```

1.65 ./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToItsFrequencyValueConverter.cs

```
using Platform.Converters;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
    ↪  IConverter<Doublet<TLink>, TLink>
    {
        private readonly LinkFrequenciesCache<TLink> _cache;
        public
        ↪  FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
        ↪  cache) => _cache = cache;
        public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
    }
}
```

1.66 ./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounte

```
using Platform.Interfaces;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
```

```
    public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
    ↪ SequenceSymbolFrequencyOneOffCounter<TLink>
    {
        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;

        public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
        ↪ ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
            : base(links, sequenceLink, symbol)
            => _markedSequenceMatcher = markedSequenceMatcher;

        public override TLink Count()
        {
            if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
            {
                return default;
            }
            return base.Count();
        }
    }
}
```

1.67 ./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs

```
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Numbers;
using Platform.Data.Sequences;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        protected readonly ILinks<TLink> _links;
        protected readonly TLink _sequenceLink;
        protected readonly TLink _symbol;
        protected TLink _total;

        public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
        ↪ TLink symbol)
        {
            _links = links;
            _sequenceLink = sequenceLink;
            _symbol = symbol;
            _total = default;
        }

        public virtual TLink Count()
        {
            if (_comparer.Compare(_total, default) > 0)
            {
                return _total;
            }
            StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
            ↪ IsElement, VisitElement);
            return _total;
        }

        private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
        ↪ _links.IsPartialPoint(x); // TODO: Use SequenceElementCreteriaMatcher instead of
        ↪ IsPartialPoint

        private bool VisitElement(TLink element)
        {
            if (_equalityComparer.Equals(element, _symbol))
            {
                _total = Arithmetic.Increment(_total);
            }
            return true;
        }
    }
}
```

1.68 ./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.

```
using Platform.Interfaces;

```

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
    {
        private readonly ILinks<TLink> _links;
        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;

        public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
        ↪  ICriterionMatcher<TLink> markedSequenceMatcher)
        {
            _links = links;
            _markedSequenceMatcher = markedSequenceMatcher;
        }

        public TLink Count(TLink argument) => new
        ↪  TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
        ↪  _markedSequenceMatcher, argument).Count();
    }
}
```

1.69 ./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffC

```
using Platform.Interfaces;
using Platform.Numbers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
    ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLink>
    {
        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;

        public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
        ↪  ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
            : base(links, symbol)
            => _markedSequenceMatcher = markedSequenceMatcher;

        protected override void CountSequenceSymbolFrequency(TLink link)
        {
            var symbolFrequencyCounter = new
            ↪  MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
            ↪  _markedSequenceMatcher, link, _symbol);
            _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
        }
    }
}
```

1.70 ./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs

```
using Platform.Interfaces;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
    {
        private readonly ILinks<TLink> _links;
        public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;
        public TLink Count(TLink symbol) => new
        ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
    }
}
```

1.71 ./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.

```
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Numbers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
```

```
12          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
13
14          protected readonly ILinks<TLink> _links;
15          protected readonly TLink _symbol;
16          protected readonly HashSet<TLink> _visits;
17          protected TLink _total;
18
19          public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
20          {
21              _links = links;
22              _symbol = symbol;
23              _visits = new HashSet<TLink>();
24              _total = default;
25          }
26
27          public TLink Count()
28          {
29              if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
30              {
31                  return _total;
32              }
33              CountCore(_symbol);
34              return _total;
35          }
36
37          private void CountCore(TLink link)
38          {
39              var any = _links.Constants.Any;
40              if (_equalityComparer.Equals(_links.Count(any, link), default))
41              {
42                  CountSequenceSymbolFrequency(link);
43              }
44              else
45              {
46                  _links.Each(EachElementHandler, any, link);
47              }
48          }
49
50          protected virtual void CountSequenceSymbolFrequency(TLink link)
51          {
52              var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
                ↪  link, _symbol);
53              _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
54          }
55
56          private TLink EachElementHandler(IList<TLink> doublet)
57          {
58              var constants = _links.Constants;
59              var doubletIndex = doublet[constants.IndexPart];
60              if (_visits.Add(doubletIndex))
61              {
62                  CountCore(doubletIndex);
63              }
64              return constants.Continue;
65          }
66      }
67  }
```

1.72 ./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs

```
1   using System.Collections.Generic;
2   using Platform.Interfaces;
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.HeightProviders
8   {
9       public class CachedSequenceHeightProvider<TLink> : LinksOperatorBase<TLink>,
        ↪  ISequenceHeightProvider<TLink>
10      {
11          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
12
13          private readonly TLink _heightPropertyMarker;
14          private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
15          private readonly IConverter<TLink> _addressToUnaryNumberConverter;
16          private readonly IConverter<TLink> _unaryNumberToAddressConverter;
17          private readonly IProperties<TLink, TLink, TLink> _propertyOperator;
18
19          public CachedSequenceHeightProvider(
20              ILinks<TLink> links,
```

```
            ISequenceHeightProvider<TLink> baseHeightProvider,
            IConverter<TLink> addressToUnaryNumberConverter,
            IConverter<TLink> unaryNumberToAddressConverter,
            TLink heightPropertyMarker,
            IProperties<TLink, TLink, TLink> propertyOperator)
            : base(links)
        {
            _heightPropertyMarker = heightPropertyMarker;
            _baseHeightProvider = baseHeightProvider;
            _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
            _propertyOperator = propertyOperator;
        }

        public TLink Get(TLink sequence)
        {
            TLink height;
            var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
            if (_equalityComparer.Equals(heightValue, default))
            {
                height = _baseHeightProvider.Get(sequence);
                heightValue = _addressToUnaryNumberConverter.Convert(height);
                _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
            }
            else
            {
                height = _unaryNumberToAddressConverter.Convert(heightValue);
            }
            return height;
        }
    }
}
```

1.73 ./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs

```
using Platform.Interfaces;
using Platform.Numbers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
    ↪  ISequenceHeightProvider<TLink>
    {
        private readonly ICriterionMatcher<TLink> _elementMatcher;

        public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
        ↪  elementMatcher) : base(links) => _elementMatcher = elementMatcher;

        public TLink Get(TLink sequence)
        {
            var height = default(TLink);
            var pairOrElement = sequence;
            while (!_elementMatcher.IsMatched(pairOrElement))
            {
                pairOrElement = Links.GetTarget(pairOrElement);
                height = Arithmetic.Increment(height);
            }
            return height;
        }
    }
}
```

1.74 ./Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs

```
using Platform.Interfaces;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    public interface ISequenceHeightProvider<TLink> : IProvider<TLink, TLink>
    {
    }
}
```

1.75 ./Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs

```
using System.Collections.Generic;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;

```

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public class CachedFrequencyIncrementingSequenceIndex<TLink> : ISequenceIndex<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly LinkFrequenciesCache<TLink> _cache;

        public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
        ↪  _cache = cache;

        public bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
            ↪  { }
            for (; i >= 1; i--)
            {
                _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
            }
            return indexed;
        }

        private bool IsIndexedWithIncrement(TLink source, TLink target)
        {
            var frequency = _cache.GetFrequency(source, target);
            if (frequency == null)
            {
                return false;
            }
            var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
            if (indexed)
            {
                _cache.IncrementFrequency(source, target);
            }
            return indexed;
        }

        public bool MightContain(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
            return indexed;
        }

        private bool IsIndexed(TLink source, TLink target)
        {
            var frequency = _cache.GetFrequency(source, target);
            if (frequency == null)
            {
                return false;
            }
            return !_equalityComparer.Equals(frequency.Frequency, default);
        }
    }
}
```

1.76 ./Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs

```
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Incrementers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public class FrequencyIncrementingSequenceIndex<TLink> : SequenceIndex<TLink>,
    ↪  ISequenceIndex<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
        private readonly IIncrementer<TLink> _frequencyIncrementer;

```

```
        public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IProperty<TLink, TLink>
        ↪  frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
            : base(links)
        {
            _frequencyPropertyOperator = frequencyPropertyOperator;
            _frequencyIncrementer = frequencyIncrementer;
        }

        public override bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
            ↪  { }
            for (; i >= 1; i--)
            {
                Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
            }
            return indexed;
        }

        private bool IsIndexedWithIncrement(TLink source, TLink target)
        {
            var link = Links.SearchOrDefault(source, target);
            var indexed = !_equalityComparer.Equals(link, default);
            if (indexed)
            {
                Increment(link);
            }
            return indexed;
        }

        private void Increment(TLink link)
        {
            var previousFrequency = _frequencyPropertyOperator.Get(link);
            var frequency = _frequencyIncrementer.Increment(previousFrequency);
            _frequencyPropertyOperator.Set(link, frequency);
        }
    }
}
```

1.77 ./Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs

```
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public interface ISequenceIndex<TLink>
    {
        /// <summary>
        /// Индексирует последовательность глобально, и возвращает значение,
        /// определяющие была ли запрошенная последовательность проиндексирована ранее.
        /// </summary>
        /// <param name="sequence">Последовательность для индексации.</param>
        bool Add(IList<TLink> sequence);

        bool MightContain(IList<TLink> sequence);
    }
}
```

1.78 ./Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs

```
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public class SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        public SequenceIndex(ILinks<TLink> links) : base(links) { }

        public virtual bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
```

```
            while (--i >= 1 && (indexed =
            ↪  !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
            ↪  default))) { }
            for (; i >= 1; i--)
            {
                Links.GetOrCreate(sequence[i - 1], sequence[i]);
            }
            return indexed;
        }

        public virtual bool MightContain(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed =
            ↪  !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
            ↪  default))) { }
            return indexed;
        }
    }
}
```

1.79 ./Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs

```
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly ISynchronizedLinks<TLink> _links;

        public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;

        public bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            var links = _links.Unsync;
            _links.SyncRoot.ExecuteReadOperation(() =>
            {
                while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                ↪  sequence[i]), default))) { }
            });
            if (!indexed)
            {
                _links.SyncRoot.ExecuteWriteOperation(() =>
                {
                    for (; i >= 1; i--)
                    {
                        links.GetOrCreate(sequence[i - 1], sequence[i]);
                    }
                });
            }
            return indexed;
        }

        public bool MightContain(IList<TLink> sequence)
        {
            var links = _links.Unsync;
            return _links.SyncRoot.ExecuteReadOperation(() =>
            {
                var indexed = true;
                var i = sequence.Count;
                while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                ↪  sequence[i]), default))) { }
                return indexed;
            });
        }
    }
}
```

1.80 ./Platform.Data.Doublets/Sequences/Indexes/Unindex.cs

```
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Indexes
{
    public class Unindex<TLink> : ISequenceIndex<TLink>
    {
        public virtual bool Add(IList<TLink> sequence) => false;

        public virtual bool MightContain(IList<TLink> sequence) => true;
    }
}
```

1.81 ./Platform.Data.Doublets/Sequences/Sequences.Experiments.cs

```
using System;
using LinkIndex = System.UInt64;
using System.Collections.Generic;
using Stack = System.Collections.Generic.Stack<ulong>;
using System.Linq;
using System.Text;
using Platform.Collections;
using Platform.Collections.Sets;
using Platform.Collections.Stacks;
using Platform.Data.Exceptions;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Walkers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences
{
    partial class Sequences
    {
        #region Create All Variants (Not Practical)

        /// <remarks>
        /// Number of links that is needed to generate all variants for
        /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
        /// </remarks>
        public ulong[] CreateAllVariants2(ulong[] sequence)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return Array.Empty<ulong>();
                }
                Links.EnsureLinkExists(sequence);
                if (sequence.Length == 1)
                {
                    return sequence;
                }
                return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
            });
        }

        private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
        {
#if DEBUG
            if ((stopAt - startAt) < 0)
            {
                throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
                ↪ меньше или равен stopAt");
            }
#endif
            if ((stopAt - startAt) == 0)
            {
                return new[] { sequence[startAt] };
            }
            if ((stopAt - startAt) == 1)
            {
                return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt]) };
            }
            var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
            var last = 0;
            for (var splitter = startAt; splitter < stopAt; splitter++)
            {
```

```
64                  var left = CreateAllVariants2Core(sequence, startAt, splitter);
65                  var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
66                  for (var i = 0; i < left.Length; i++)
67                  {
68                      for (var j = 0; j < right.Length; j++)
69                      {
70                          var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
71                          if (variant == Constants.Null)
72                          {
73                              throw new NotImplementedException("Creation cancellation is not
                                ↪  implemented.");
74                          }
75                          variants[last++] = variant;
76                      }
77                  }
78              }
79              return variants;
80          }
81
82          public List<ulong> CreateAllVariants1(params ulong[] sequence)
83          {
84              return _sync.ExecuteWriteOperation(() =>
85              {
86                  if (sequence.IsNullOrEmpty())
87                  {
88                      return new List<ulong>();
89                  }
90                  Links.Unsync.EnsureLinkExists(sequence);
91                  if (sequence.Length == 1)
92                  {
93                      return new List<ulong> { sequence[0] };
94                  }
95                  var results = new
                    ↪  List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
96                  return CreateAllVariants1Core(sequence, results);
97              });
98          }
99
100         private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
101         {
102             if (sequence.Length == 2)
103             {
104                 var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
105                 if (link == Constants.Null)
106                 {
107                     throw new NotImplementedException("Creation cancellation is not
                        ↪  implemented.");
108                 }
109                 results.Add(link);
110                 return results;
111             }
112             var innerSequenceLength = sequence.Length - 1;
113             var innerSequence = new ulong[innerSequenceLength];
114             for (var li = 0; li < innerSequenceLength; li++)
115             {
116                 var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
117                 if (link == Constants.Null)
118                 {
119                     throw new NotImplementedException("Creation cancellation is not
                        ↪  implemented.");
120                 }
121                 for (var isi = 0; isi < li; isi++)
122                 {
123                     innerSequence[isi] = sequence[isi];
124                 }
125                 innerSequence[li] = link;
126                 for (var isi = li + 1; isi < innerSequenceLength; isi++)
127                 {
128                     innerSequence[isi] = sequence[isi + 1];
129                 }
130                 CreateAllVariants1Core(innerSequence, results);
131             }
132             return results;
133         }
134
135         #endregion
136
137         public HashSet<ulong> Each1(params ulong[] sequence)
```

```
        {
            var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
            Each1(link =>
            {
                if (!visitedLinks.Contains(link))
                {
                    visitedLinks.Add(link); // изучить почему случаются повторы
                }
                return true;
            }, sequence);
            return visitedLinks;
        }

        private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
        {
            if (sequence.Length == 2)
            {
                Links.Unsync.Each(sequence[0], sequence[1], handler);
            }
            else
            {
                var innerSequenceLength = sequence.Length - 1;
                for (var li = 0; li < innerSequenceLength; li++)
                {
                    var left = sequence[li];
                    var right = sequence[li + 1];
                    if (left == 0 && right == 0)
                    {
                        continue;
                    }
                    var linkIndex = li;
                    ulong[] innerSequence = null;
                    Links.Unsync.Each(doublet =>
                    {
                        if (innerSequence == null)
                        {
                            innerSequence = new ulong[innerSequenceLength];
                            for (var isi = 0; isi < linkIndex; isi++)
                            {
                                innerSequence[isi] = sequence[isi];
                            }
                            for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
                            {
                                innerSequence[isi] = sequence[isi + 1];
                            }
                        }
                        innerSequence[linkIndex] = doublet[Constants.IndexPart];
                        Each1(handler, innerSequence);
                        return Constants.Continue;
                    }, Constants.Any, left, right);
                }
            }
        }

        public HashSet<ulong> EachPart(params ulong[] sequence)
        {
            var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
            EachPartCore(link =>
            {
                var linkIndex = link[Constants.IndexPart];
                if (!visitedLinks.Contains(linkIndex))
                {
                    visitedLinks.Add(linkIndex); // изучить почему случаются повторы
                }
                return Constants.Continue;
            }, sequence);
            return visitedLinks;
        }

        public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
        {
            var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
            EachPartCore(link =>
            {
                var linkIndex = link[Constants.IndexPart];
                if (!visitedLinks.Contains(linkIndex))
                {
                    visitedLinks.Add(linkIndex); // изучить почему случаются повторы
                    return handler(new LinkAddress<LinkIndex>(linkIndex));
```

```
217                 }
218                 return Constants.Continue;
219             }, sequence);
220         }
221 
222         private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
    ↪  sequence)
223         {
224             if (sequence.IsNullOrEmpty())
225             {
226                 return;
227             }
228             Links.EnsureLinkIsAnyOrExists(sequence);
229             if (sequence.Length == 1)
230             {
231                 var link = sequence[0];
232                 if (link > 0)
233                 {
234                     handler(new LinkAddress<LinkIndex>(link));
235                 }
236                 else
237                 {
238                     Links.Each(Constants.Any, Constants.Any, handler);
239                 }
240             }
241             else if (sequence.Length == 2)
242             {
243                 //_links.Each(sequence[0], sequence[1], handler);
244                 //  o_|      x_o ...
245                 // x_|        |___|
246                 Links.Each(sequence[1], Constants.Any, doublet =>
247                 {
248                     var match = Links.SearchOrDefault(sequence[0], doublet);
249                     if (match != Constants.Null)
250                     {
251                         handler(new LinkAddress<LinkIndex>(match));
252                     }
253                     return true;
254                 });
255                 // |_x      ... x_o
256                 //  |_o      |___|
257                 Links.Each(Constants.Any, sequence[0], doublet =>
258                 {
259                     var match = Links.SearchOrDefault(doublet, sequence[1]);
260                     if (match != 0)
261                     {
262                         handler(new LinkAddress<LinkIndex>(match));
263                     }
264                     return true;
265                 });
266                 //           ._x o_.
267                 //            |___|
268                 PartialStepRight(x => handler(x), sequence[0], sequence[1]);
269             }
270             else
271             {
272                 throw new NotImplementedException();
273             }
274         }
275 
276         private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
277         {
278             Links.Unsync.Each(Constants.Any, left, doublet =>
279             {
280                 StepRight(handler, doublet, right);
281                 if (left != doublet)
282                 {
283                     PartialStepRight(handler, doublet, right);
284                 }
285                 return true;
286             });
287         }
288 
289         private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
290         {
291             Links.Unsync.Each(left, Constants.Any, rightStep =>
292             {
293                 TryStepRightUp(handler, right, rightStep);
```

```
294                 return true;
295             });
296         }
297 
298         private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
        ↪  stepFrom)
299         {
300             var upStep = stepFrom;
301             var firstSource = Links.Unsync.GetTarget(upStep);
302             while (firstSource != right && firstSource != upStep)
303             {
304                 upStep = firstSource;
305                 firstSource = Links.Unsync.GetSource(upStep);
306             }
307             if (firstSource == right)
308             {
309                 handler(new LinkAddress<LinkIndex>(stepFrom));
310             }
311         }
312 
313         // TODO: Test
314         private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
315         {
316             Links.Unsync.Each(right, Constants.Any, doublet =>
317             {
318                 StepLeft(handler, left, doublet);
319                 if (right != doublet)
320                 {
321                     PartialStepLeft(handler, left, doublet);
322                 }
323                 return true;
324             });
325         }
326 
327         private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
328         {
329             Links.Unsync.Each(Constants.Any, right, leftStep =>
330             {
331                 TryStepLeftUp(handler, left, leftStep);
332                 return true;
333             });
334         }
335 
336         private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
337         {
338             var upStep = stepFrom;
339             var firstTarget = Links.Unsync.GetSource(upStep);
340             while (firstTarget != left && firstTarget != upStep)
341             {
342                 upStep = firstTarget;
343                 firstTarget = Links.Unsync.GetTarget(upStep);
344             }
345             if (firstTarget == left)
346             {
347                 handler(new LinkAddress<LinkIndex>(stepFrom));
348             }
349         }
350 
351         private bool StartsWith(ulong sequence, ulong link)
352         {
353             var upStep = sequence;
354             var firstSource = Links.Unsync.GetSource(upStep);
355             while (firstSource != link && firstSource != upStep)
356             {
357                 upStep = firstSource;
358                 firstSource = Links.Unsync.GetSource(upStep);
359             }
360             return firstSource == link;
361         }
362 
363         private bool EndsWith(ulong sequence, ulong link)
364         {
365             var upStep = sequence;
366             var lastTarget = Links.Unsync.GetTarget(upStep);
367             while (lastTarget != link && lastTarget != upStep)
368             {
369                 upStep = lastTarget;
370                 lastTarget = Links.Unsync.GetTarget(upStep);
371             }
```

```
372             return lastTarget == link;
373         }
374 
375         public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
376         {
377             return _sync.ExecuteReadOperation(() =>
378             {
379                 var results = new List<ulong>();
380                 if (sequence.Length > 0)
381                 {
382                     Links.EnsureLinkExists(sequence);
383                     var firstElement = sequence[0];
384                     if (sequence.Length == 1)
385                     {
386                         results.Add(firstElement);
387                         return results;
388                     }
389                     if (sequence.Length == 2)
390                     {
391                         var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
392                         if (doublet != Constants.Null)
393                         {
394                             results.Add(doublet);
395                         }
396                         return results;
397                     }
398                     var linksInSequence = new HashSet<ulong>(sequence);
399                     void handler(IList<LinkIndex> result)
400                     {
401                         var resultIndex = result[Links.Constants.IndexPart];
402                         var filterPosition = 0;
403                         StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
                            ↪  Links.Unsync.GetTarget,
404                             x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                                ↪  x =>
405                             {
406                                 if (filterPosition == sequence.Length)
407                                 {
408                                     filterPosition = -2; // Длиннее чем нужно
409                                     return false;
410                                 }
411                                 if (x != sequence[filterPosition])
412                                 {
413                                     filterPosition = -1;
414                                     return false; // Начинается иначе
415                                 }
416                                 filterPosition++;
417 
418                                 return true;
419                             });
420                         if (filterPosition == sequence.Length)
421                         {
422                             results.Add(resultIndex);
423                         }
424                     }
425                     if (sequence.Length >= 2)
426                     {
427                         StepRight(handler, sequence[0], sequence[1]);
428                     }
429                     var last = sequence.Length - 2;
430                     for (var i = 1; i < last; i++)
431                     {
432                         PartialStepRight(handler, sequence[i], sequence[i + 1]);
433                     }
434                     if (sequence.Length >= 3)
435                     {
436                         StepLeft(handler, sequence[sequence.Length - 2],
                            ↪  sequence[sequence.Length - 1]);
437                     }
438                 }
439                 return results;
440             });
441         }
442 
443         public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
444         {
445             return _sync.ExecuteReadOperation(() =>
446             {
447                 var results = new HashSet<ulong>();
```

```
448                 if (sequence.Length > 0)
449                 {
450                     Links.EnsureLinkExists(sequence);
451                     var firstElement = sequence[0];
452                     if (sequence.Length == 1)
453                     {
454                         results.Add(firstElement);
455                         return results;
456                     }
457                     if (sequence.Length == 2)
458                     {
459                         var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
460                         if (doublet != Constants.Null)
461                         {
462                             results.Add(doublet);
463                         }
464                         return results;
465                     }
466                     var matcher = new Matcher(this, sequence, results, null);
467                     if (sequence.Length >= 2)
468                     {
469                         StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
470                     }
471                     var last = sequence.Length - 2;
472                     for (var i = 1; i < last; i++)
473                     {
474                         PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
                             ↪ sequence[i + 1]);
475                     }
476                     if (sequence.Length >= 3)
477                     {
478                         StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
                             ↪ sequence[sequence.Length - 1]);
479                     }
480                 }
481                 return results;
482             });
483         }
484 
485         public const int MaxSequenceFormatSize = 200;
486 
487         public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
            ↪ => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
488 
489         public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
            ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
            ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
            ↪ elementToString, insertComma, knownElements));
490 
491         private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
            ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
            ↪ LinkIndex[] knownElements)
492         {
493             var linksInSequence = new HashSet<ulong>(knownElements);
494             //var entered = new HashSet<ulong>();
495             var sb = new StringBuilder();
496             sb.Append('{');
497             if (links.Exists(sequenceLink))
498             {
499                 StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
500                     x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
                         ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
501                     {
502                         if (insertComma && sb.Length > 1)
503                         {
504                             sb.Append(',');
505                         }
506                         //if (entered.Contains(element))
507                         //{
508                         //    sb.Append('{');
509                         //    elementToString(sb, element);
510                         //    sb.Append('}');
511                         //}
512                         //else
513                         elementToString(sb, element);
514                         if (sb.Length < MaxSequenceFormatSize)
515                         {
516                             return true;
```

```
                    }
                    sb.Append(insertComma ? ", ..." : "...");
                    return false;
                });
        }
        sb.Append('}');
        return sb.ToString();
    }

    public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
    ↪ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
    ↪ knownElements);

    public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
    ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    ↪ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
    ↪ sequenceLink, elementToString, insertComma, knownElements));

    private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
    ↪ LinkIndex[] knownElements)
    {
        var linksInSequence = new HashSet<ulong>(knownElements);
        var entered = new HashSet<ulong>();
        var sb = new StringBuilder();
        sb.Append('{');
        if (links.Exists(sequenceLink))
        {
            StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
                x => linksInSequence.Contains(x) || links.IsFullPoint(x),
                ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
                {
                    if (insertComma && sb.Length > 1)
                    {
                        sb.Append(',');
                    }
                    if (entered.Contains(element))
                    {
                        sb.Append('{');
                        elementToString(sb, element);
                        sb.Append('}');
                    }
                    else
                    {
                        elementToString(sb, element);
                    }
                    if (sb.Length < MaxSequenceFormatSize)
                    {
                        return true;
                    }
                    sb.Append(insertComma ? ", ..." : "...");
                    return false;
                });
        }
        sb.Append('}');
        return sb.ToString();
    }

    public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            if (sequence.Length > 0)
            {
                Links.EnsureLinkExists(sequence);
                var results = new HashSet<ulong>();
                for (var i = 0; i < sequence.Length; i++)
                {
                    AllUsagesCore(sequence[i], results);
                }
                var filteredResults = new List<ulong>();
                var linksInSequence = new HashSet<ulong>(sequence);
                foreach (var result in results)
                {
                    var filterPosition = -1;
                    StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
                    ↪  Links.Unsync.GetTarget,
```

```
                            x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                            ↪  x =>
                            {
                                if (filterPosition == (sequence.Length - 1))
                                {
                                    return false;
                                }
                                if (filterPosition >= 0)
                                {
                                    if (x == sequence[filterPosition + 1])
                                    {
                                        filterPosition++;
                                    }
                                    else
                                    {
                                        return false;
                                    }
                                }
                                if (filterPosition < 0)
                                {
                                    if (x == sequence[0])
                                    {
                                        filterPosition = 0;
                                    }
                                }
                                return true;
                            });
                        if (filterPosition == (sequence.Length - 1))
                        {
                            filteredResults.Add(result);
                        }
                    }
                    return filteredResults;
                }
                return new List<ulong>();
            });
        }

        public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(sequence);
                    var results = new HashSet<ulong>();
                    for (var i = 0; i < sequence.Length; i++)
                    {
                        AllUsagesCore(sequence[i], results);
                    }
                    var filteredResults = new HashSet<ulong>();
                    var matcher = new Matcher(this, sequence, filteredResults, null);
                    matcher.AddAllPartialMatchedToResults(results);
                    return filteredResults;
                }
                return new HashSet<ulong>();
            });
        }

        public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
        ↪  params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(sequence);

                    var results = new HashSet<ulong>();
                    var filteredResults = new HashSet<ulong>();
                    var matcher = new Matcher(this, sequence, filteredResults, handler);
                    for (var i = 0; i < sequence.Length; i++)
                    {
                        if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
                        {
                            return false;
                        }
                    }
                    return true;
```

```
662                 }
663                 return true;
664             });
665         }
666 
667         //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
668         //{
669         //    return Sync.ExecuteReadOperation(() =>
670         //    {
671         //        if (sequence.Length > 0)
672         //        {
673         //            _links.EnsureEachLinkIsAnyOrExists(sequence);
674 
675         //            var firstResults = new HashSet<ulong>();
676         //            var lastResults = new HashSet<ulong>();
677 
678         //            var first = sequence.First(x => x != LinksConstants.Any);
679         //            var last = sequence.Last(x => x != LinksConstants.Any);
680 
681         //            AllUsagesCore(first, firstResults);
682         //            AllUsagesCore(last, lastResults);
683 
684         //            firstResults.IntersectWith(lastResults);
685 
686         //            //for (var i = 0; i < sequence.Length; i++)
687         //            //    AllUsagesCore(sequence[i], results);
688 
689         //            var filteredResults = new HashSet<ulong>();
690         //            var matcher = new Matcher(this, sequence, filteredResults, null);
691         //            matcher.AddAllPartialMatchedToResults(firstResults);
692         //            return filteredResults;
693         //        }
694 
695         //        return new HashSet<ulong>();
696         //    });
697         //}
698 
699         public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
700         {
701             return _sync.ExecuteReadOperation((Func<HashSet<ulong>>)(() =>
702             {
703                 if (sequence.Length > 0)
704                 {
705                     ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links,
                        ↪  (IList<ulong>)sequence);
706                     var firstResults = new HashSet<ulong>();
707                     var lastResults = new HashSet<ulong>();
708                     var first = sequence.First(x => x != Constants.Any);
709                     var last = sequence.Last(x => x != Constants.Any);
710                     AllUsagesCore(first, firstResults);
711                     AllUsagesCore(last, lastResults);
712                     firstResults.IntersectWith(lastResults);
713                     //for (var i = 0; i < sequence.Length; i++)
714                     //    AllUsagesCore(sequence[i], results);
715                     var filteredResults = new HashSet<ulong>();
716                     var matcher = new Matcher(this, sequence, filteredResults, null);
717                     matcher.AddAllPartialMatchedToResults(firstResults);
718                     return filteredResults;
719                 }
720                 return new HashSet<ulong>();
721             }));
722         }
723 
724         public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
            ↪  IList<ulong> sequence)
725         {
726             return _sync.ExecuteReadOperation(() =>
727             {
728                 if (sequence.Count > 0)
729                 {
730                     Links.EnsureLinkExists(sequence);
731                     var results = new HashSet<LinkIndex>();
732                     //var nextResults = new HashSet<ulong>();
733                     //for (var i = 0; i < sequence.Length; i++)
734                     //{
735                     //    AllUsagesCore(sequence[i], nextResults);
736                     //    if (results.IsNullOrEmpty())
737                     //    {
```

```
                    //        results = nextResults;
                    //        nextResults = new HashSet<ulong>();
                    //    }
                    //    else
                    //    {
                    //        results.IntersectWith(nextResults);
                    //        nextResults.Clear();
                    //    }
                    //}
                    var collector1 = new AllUsagesCollector1(Links.Unsync, results);
                    collector1.Collect(Links.Unsync.GetLink(sequence[0]));
                    var next = new HashSet<ulong>();
                    for (var i = 1; i < sequence.Count; i++)
                    {
                        var collector = new AllUsagesCollector1(Links.Unsync, next);
                        collector.Collect(Links.Unsync.GetLink(sequence[i]));

                        results.IntersectWith(next);
                        next.Clear();
                    }
                    var filteredResults = new HashSet<ulong>();
                    var matcher = new Matcher(this, sequence, filteredResults, null,
                    ↪  readAsElements);
                    matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
                    ↪  x)); // OrderBy is a Hack
                    return filteredResults;
                }
                return new HashSet<ulong>();
            });
        }

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

        public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(sequence);
                    //var firstElement = sequence[0];
                    //if (sequence.Length == 1)
                    //{
                    //    //results.Add(firstElement);
                    //    return results;
                    //}
                    //if (sequence.Length == 2)
                    //{
                    //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
                    //    //if (doublet != Doublets.Links.Null)
                    //    //    results.Add(doublet);
                    //    return results;
                    //}
                    //var lastElement = sequence[sequence.Length - 1];
                    //Func<ulong, bool> handler = x =>
                    //{
                    //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
                    ↪  results.Add(x);
                    //    return true;
                    //};
                    //if (sequence.Length >= 2)
                    //    StepRight(handler, sequence[0], sequence[1]);
                    //var last = sequence.Length - 2;
```

```
                    //for (var i = 1; i < last; i++)
                    //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
                    //if (sequence.Length >= 3)
                    //    StepLeft(handler, sequence[sequence.Length - 2],
                    ↪  sequence[sequence.Length - 1]);
                    //////if (sequence.Length == 1)
                    //////{
                    //////    throw new NotImplementedException(); // all sequences, containing
                    ↪  this element?
                    //////}
                    //////if (sequence.Length == 2)
                    //////{
                    //////    var results = new List<ulong>();
                    //////    PartialStepRight(results.Add, sequence[0], sequence[1]);
                    //////    return results;
                    //////}
                    //////var matches = new List<List<ulong>>();
                    //////var last = sequence.Length - 1;
                    //////for (var i = 0; i < last; i++)
                    //////{
                    //////    var results = new List<ulong>();
                    //////    //StepRight(results.Add, sequence[i], sequence[i + 1]);
                    //////    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
                    //////    if (results.Count > 0)
                    //////        matches.Add(results);
                    //////    else
                    //////        return results;
                    //////    if (matches.Count == 2)
                    //////    {
                    //////        var merged = new List<ulong>();
                    //////        for (var j = 0; j < matches[0].Count; j++)
                    //////            for (var k = 0; k < matches[1].Count; k++)
                    //////                CloseInnerConnections(merged.Add, matches[0][j],
                    ↪  matches[1][k]);
                    //////        if (merged.Count > 0)
                    //////            matches = new List<List<ulong>> { merged };
                    //////        else
                    //////            return new List<ulong>();
                    //////    }
                    //////}
                    //////if (matches.Count > 0)
                    //////{
                    //////    var usages = new HashSet<ulong>();
                    //////    for (int i = 0; i < sequence.Length; i++)
                    //////    {
                    //////        AllUsagesCore(sequence[i], usages);
                    //////    }
                    //////    //for (int i = 0; i < matches[0].Count; i++)
                    //////    //    AllUsagesCore(matches[0][i], usages);
                    //////    //usages.UnionWith(matches[0]);
                    //////    return usages.ToList();
                    //////}
                    var firstLinkUsages = new HashSet<ulong>();
                    AllUsagesCore(sequence[0], firstLinkUsages);
                    firstLinkUsages.Add(sequence[0]);
                    //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
                    ↪  sequence[0] }; // or all sequences, containing this element?
                    //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
                    ↪  1).ToList();
                    var results = new HashSet<ulong>();
                    foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
                    ↪  firstLinkUsages, 1))
                    {
                        AllUsagesCore(match, results);
                    }
                    return results.ToList();
                }
                return new List<ulong>();
            });
        }

        /// <remarks>
        /// TODO: Может потробоваться ограничение на уровень глубины рекурсии
        /// </remarks>
        public HashSet<ulong> AllUsages(ulong link)
        {
            return _sync.ExecuteReadOperation(() =>
```

```
            {
                var usages = new HashSet<ulong>();
                AllUsagesCore(link, usages);
                return usages;
            });
        }

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

        public HashSet<ulong> AllBottomUsages(ulong link)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var visits = new HashSet<ulong>();
                var usages = new HashSet<ulong>();
                AllBottomUsagesCore(link, visits, usages);
                return usages;
            });
        }

        private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong> usages)
        {
            bool handler(ulong doublet)
            {
                if (visits.Add(doublet))
                {
                    AllBottomUsagesCore(doublet, visits, usages);
                }
                return true;
            }
            if (Links.Unsync.Count(Constants.Any, link) == 0)
            {
                usages.Add(link);
            }
            else
            {
                Links.Unsync.Each(link, Constants.Any, handler);
                Links.Unsync.Each(Constants.Any, link, handler);
            }
        }

        public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
        {
            if (Options.UseSequenceMarker)
            {
                var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links, Options.MarkedSequenceMatcher, symbol);
                return counter.Count();
            }
            else
            {
                var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links, symbol);
                return counter.Count();
            }
        }

        private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>, LinkIndex> outerHandler)
        {
            bool handler(ulong doublet)
            {
```

```
                if (usages.Add(doublet))
                {
                    if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
                    {
                        return false;
                    }
                    if (!AllUsagesCore1(doublet, usages, outerHandler))
                    {
                        return false;
                    }
                }
                return true;
            }
            return Links.Unsync.Each(link, Constants.Any, handler)
                && Links.Unsync.Each(Constants.Any, link, handler);
        }

        public void CalculateAllUsages(ulong[] totals)
        {
            var calculator = new AllUsagesCalculator(Links, totals);
            calculator.Calculate();
        }

        public void CalculateAllUsages2(ulong[] totals)
        {
            var calculator = new AllUsagesCalculator2(Links, totals);
            calculator.Calculate();
        }

        private class AllUsagesCalculator
        {
            private readonly SynchronizedLinks<ulong> _links;
            private readonly ulong[] _totals;

            public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
            {
                _links = links;
                _totals = totals;
            }

            public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
            ↪  CalculateCore);

            private bool CalculateCore(ulong link)
            {
                if (_totals[link] == 0)
                {
                    var total = 1UL;
                    _totals[link] = total;
                    var visitedChildren = new HashSet<ulong>();
                    bool linkCalculator(ulong child)
                    {
                        if (link != child && visitedChildren.Add(child))
                        {
                            total += _totals[child] == 0 ? 1 : _totals[child];
                        }
                        return true;
                    }
                    _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
                    _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
                    _totals[link] = total;
                }
                return true;
            }
        }

        private class AllUsagesCalculator2
        {
            private readonly SynchronizedLinks<ulong> _links;
            private readonly ulong[] _totals;

            public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
            {
                _links = links;
                _totals = totals;
            }

            public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
            ↪  CalculateCore);
```

```
1031
1032            private bool IsElement(ulong link)
1033            {
1034                //_linksInSequence.Contains(link) ||
1035                return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
                    ↪ link;
1036            }
1037
1038            private bool CalculateCore(ulong link)
1039            {
1040                // TODO: Проработать защиту от зацикливания
1041                // Основано на SequenceWalker.WalkLeft
1042                Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1043                Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1044                Func<ulong, bool> isElement = IsElement;
1045                void visitLeaf(ulong parent)
1046                {
1047                    if (link != parent)
1048                    {
1049                        _totals[parent]++;
1050                    }
1051                }
1052                void visitNode(ulong parent)
1053                {
1054                    if (link != parent)
1055                    {
1056                        _totals[parent]++;
1057                    }
1058                }
1059                var stack = new Stack();
1060                var element = link;
1061                if (isElement(element))
1062                {
1063                    visitLeaf(element);
1064                }
1065                else
1066                {
1067                    while (true)
1068                    {
1069                        if (isElement(element))
1070                        {
1071                            if (stack.Count == 0)
1072                            {
1073                                break;
1074                            }
1075                            element = stack.Pop();
1076                            var source = getSource(element);
1077                            var target = getTarget(element);
1078                            // Обработка элемента
1079                            if (isElement(target))
1080                            {
1081                                visitLeaf(target);
1082                            }
1083                            if (isElement(source))
1084                            {
1085                                visitLeaf(source);
1086                            }
1087                            element = source;
1088                        }
1089                        else
1090                        {
1091                            stack.Push(element);
1092                            visitNode(element);
1093                            element = getTarget(element);
1094                        }
1095                    }
1096                }
1097                _totals[link]++;
1098                return true;
1099            }
1100        }
1101
1102        private class AllUsagesCollector
1103        {
1104            private readonly ILinks<ulong> _links;
1105            private readonly HashSet<ulong> _usages;
1106
1107            public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1108            {
```

```
1109                 _links = links;
1110                 _usages = usages;
1111             }
1112 
1113             public bool Collect(ulong link)
1114             {
1115                 if (_usages.Add(link))
1116                 {
1117                     _links.Each(link, _links.Constants.Any, Collect);
1118                     _links.Each(_links.Constants.Any, link, Collect);
1119                 }
1120                 return true;
1121             }
1122         }
1123 
1124         private class AllUsagesCollector1
1125         {
1126             private readonly ILinks<ulong> _links;
1127             private readonly HashSet<ulong> _usages;
1128             private readonly ulong _continue;
1129 
1130             public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1131             {
1132                 _links = links;
1133                 _usages = usages;
1134                 _continue = _links.Constants.Continue;
1135             }
1136 
1137             public ulong Collect(IList<ulong> link)
1138             {
1139                 var linkIndex = _links.GetIndex(link);
1140                 if (_usages.Add(linkIndex))
1141                 {
1142                     _links.Each(Collect, _links.Constants.Any, linkIndex);
1143                 }
1144                 return _continue;
1145             }
1146         }
1147 
1148         private class AllUsagesCollector2
1149         {
1150             private readonly ILinks<ulong> _links;
1151             private readonly BitString _usages;
1152 
1153             public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1154             {
1155                 _links = links;
1156                 _usages = usages;
1157             }
1158 
1159             public bool Collect(ulong link)
1160             {
1161                 if (_usages.Add((long)link))
1162                 {
1163                     _links.Each(link, _links.Constants.Any, Collect);
1164                     _links.Each(_links.Constants.Any, link, Collect);
1165                 }
1166                 return true;
1167             }
1168         }
1169 
1170         private class AllUsagesIntersectingCollector
1171         {
1172             private readonly SynchronizedLinks<ulong> _links;
1173             private readonly HashSet<ulong> _intersectWith;
1174             private readonly HashSet<ulong> _usages;
1175             private readonly HashSet<ulong> _enter;
1176 
1177             public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
                 ↪  intersectWith, HashSet<ulong> usages)
1178             {
1179                 _links = links;
1180                 _intersectWith = intersectWith;
1181                 _usages = usages;
1182                 _enter = new HashSet<ulong>(); // защита от зацикливания
1183             }
1184 
1185             public bool Collect(ulong link)
1186             {
1187                 if (_enter.Add(link))
```

```
            {
                if (_intersectWith.Contains(link))
                {
                    _usages.Add(link);
                }
                _links.Unsync.Each(link, _links.Constants.Any, Collect);
                _links.Unsync.Each(_links.Constants.Any, link, Collect);
            }
            return true;
        }
    }

    private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
    ↪ right)
    {
        TryStepLeftUp(handler, left, right);
        TryStepRightUp(handler, right, left);
    }

    private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
    ↪ right)
    {
        // Direct
        if (left == right)
        {
            handler(new LinkAddress<LinkIndex>(left));
        }
        var doublet = Links.Unsync.SearchOrDefault(left, right);
        if (doublet != Constants.Null)
        {
            handler(new LinkAddress<LinkIndex>(doublet));
        }
        // Inner
        CloseInnerConnections(handler, left, right);
        // Outer
        StepLeft(handler, left, right);
        StepRight(handler, left, right);
        PartialStepRight(handler, left, right);
        PartialStepLeft(handler, left, right);
    }

    private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
    ↪ HashSet<ulong> previousMatchings, long startAt)
    {
        if (startAt >= sequence.Length) // ?
        {
            return previousMatchings;
        }
        var secondLinkUsages = new HashSet<ulong>();
        AllUsagesCore(sequence[startAt], secondLinkUsages);
        secondLinkUsages.Add(sequence[startAt]);
        var matchings = new HashSet<ulong>();
        var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
        //for (var i = 0; i < previousMatchings.Count; i++)
        foreach (var secondLinkUsage in secondLinkUsages)
        {
            foreach (var previousMatching in previousMatchings)
            {
                //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
                ↪ secondLinkUsage);
                StepRight(filler.AddFirstAndReturnConstant, previousMatching,
                ↪ secondLinkUsage);
                TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
                ↪ previousMatching);
                //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
                ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
                ↪ желаемым результам.
                PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
                ↪ secondLinkUsage);
            }
        }
        if (matchings.Count == 0)
        {
            return matchings;
        }
        return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
    }
```

```
1256
1257        private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
            ↪ links, params ulong[] sequence)
1258        {
1259            if (sequence == null)
1260            {
1261                return;
1262            }
1263            for (var i = 0; i < sequence.Length; i++)
1264            {
1265                if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
                    ↪ !links.Exists(sequence[i]))
1266                {
1267                    throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                        ↪ $"patternSequence[{i}]");
1268                }
1269            }
1270        }
1271
1272        // Pattern Matching -> Key To Triggers
1273        public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1274        {
1275            return _sync.ExecuteReadOperation(() =>
1276            {
1277                patternSequence = Simplify(patternSequence);
1278                if (patternSequence.Length > 0)
1279                {
1280                    EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1281                    var uniqueSequenceElements = new HashSet<ulong>();
1282                    for (var i = 0; i < patternSequence.Length; i++)
1283                    {
1284                        if (patternSequence[i] != Constants.Any && patternSequence[i] !=
                            ↪ ZeroOrMany)
1285                        {
1286                            uniqueSequenceElements.Add(patternSequence[i]);
1287                        }
1288                    }
1289                    var results = new HashSet<ulong>();
1290                    foreach (var uniqueSequenceElement in uniqueSequenceElements)
1291                    {
1292                        AllUsagesCore(uniqueSequenceElement, results);
1293                    }
1294                    var filteredResults = new HashSet<ulong>();
1295                    var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1296                    matcher.AddAllPatternMatchedToResults(results);
1297                    return filteredResults;
1298                }
1299                return new HashSet<ulong>();
1300            });
1301        }
1302
1303        // Найти все возможные связи между указанным списком связей.
1304        // Находит связи между всеми указанными связями в любом порядке.
1305        // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
            ↪ несколько раз в последовательности)
1306        public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1307        {
1308            return _sync.ExecuteReadOperation(() =>
1309            {
1310                var results = new HashSet<ulong>();
1311                if (linksToConnect.Length > 0)
1312                {
1313                    Links.EnsureLinkExists(linksToConnect);
1314                    AllUsagesCore(linksToConnect[0], results);
1315                    for (var i = 1; i < linksToConnect.Length; i++)
1316                    {
1317                        var next = new HashSet<ulong>();
1318                        AllUsagesCore(linksToConnect[i], next);
1319                        results.IntersectWith(next);
1320                    }
1321                }
1322                return results;
1323            });
1324        }
1325
1326        public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1327        {
1328            return _sync.ExecuteReadOperation(() =>
```

```
            {
                var results = new HashSet<ulong>();
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureLinkExists(linksToConnect);
                    var collector1 = new AllUsagesCollector(Links.Unsync, results);
                    collector1.Collect(linksToConnect[0]);
                    var next = new HashSet<ulong>();
                    for (var i = 1; i < linksToConnect.Length; i++)
                    {
                        var collector = new AllUsagesCollector(Links.Unsync, next);
                        collector.Collect(linksToConnect[i]);
                        results.IntersectWith(next);
                        next.Clear();
                    }
                }
                return results;
            });
        }

        public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new HashSet<ulong>();
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureLinkExists(linksToConnect);
                    var collector1 = new AllUsagesCollector(Links, results);
                    collector1.Collect(linksToConnect[0]);
                    //AllUsagesCore(linksToConnect[0], results);
                    for (var i = 1; i < linksToConnect.Length; i++)
                    {
                        var next = new HashSet<ulong>();
                        var collector = new AllUsagesIntersectingCollector(Links, results, next);
                        collector.Collect(linksToConnect[i]);
                        //AllUsagesCore(linksToConnect[i], next);
                        //results.IntersectWith(next);
                        results = next;
                    }
                }
                return results;
            });
        }

        public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new BitString((long)Links.Unsync.Count() + 1); // new
                ↪  BitArray((int)_links.Total + 1);
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureLinkExists(linksToConnect);
                    var collector1 = new AllUsagesCollector2(Links.Unsync, results);
                    collector1.Collect(linksToConnect[0]);
                    for (var i = 1; i < linksToConnect.Length; i++)
                    {
                        var next = new BitString((long)Links.Unsync.Count() + 1); //new
                        ↪  BitArray((int)_links.Total + 1);
                        var collector = new AllUsagesCollector2(Links.Unsync, next);
                        collector.Collect(linksToConnect[i]);
                        results = results.And(next);
                    }
                }
                return results.GetSetUInt64Indices();
            });
        }

        private static ulong[] Simplify(ulong[] sequence)
        {
            // Считаем новый размер последовательности
            long newLength = 0;
            var zeroOrManyStepped = false;
            for (var i = 0; i < sequence.Length; i++)
            {
                if (sequence[i] == ZeroOrMany)
                {
```

```
                if (zeroOrManyStepped)
                {
                    continue;
                }
                zeroOrManyStepped = true;
            }
            else
            {
                //if (zeroOrManyStepped) Is it efficient?
                zeroOrManyStepped = false;
            }
            newLength++;
        }
        // Строим новую последовательность
        zeroOrManyStepped = false;
        var newSequence = new ulong[newLength];
        long j = 0;
        for (var i = 0; i < sequence.Length; i++)
        {
            //var current = zeroOrManyStepped;
            //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
            //if (current && zeroOrManyStepped)
            //    continue;
            //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
            //if (zeroOrManyStepped && newZeroOrManyStepped)
            //    continue;
            //zeroOrManyStepped = newZeroOrManyStepped;
            if (sequence[i] == ZeroOrMany)
            {
                if (zeroOrManyStepped)
                {
                    continue;
                }
                zeroOrManyStepped = true;
            }
            else
            {
                //if (zeroOrManyStepped) Is it efficient?
                zeroOrManyStepped = false;
            }
            newSequence[j++] = sequence[i];
        }
        return newSequence;
    }

    public static void TestSimplify()
    {
        var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
        ↪  ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
        var simplifiedSequence = Simplify(sequence);
    }

    public List<ulong> GetSimilarSequences() => new List<ulong>();

    public void Prediction()
    {
        //_links
        //sequences
    }

    #region From Triplets

    //public static void DeleteSequence(Link sequence)
    //{
    //}

    public List<ulong> CollectMatchingSequences(ulong[] links)
    {
        if (links.Length == 1)
        {
            throw new Exception("Подпоследовательности с одним элементом не
            ↪  поддерживаются.");
        }
        var leftBound = 0;
        var rightBound = links.Length - 1;
        var left = links[leftBound++];
        var right = links[rightBound--];
        var results = new List<ulong>();
        CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
```

```
1482            return results;
1483        }
1484
1485        private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
            ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1486        {
1487            var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1488            var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1489            if (leftLinkTotalReferers <= rightLinkTotalReferers)
1490            {
1491                var nextLeftLink = middleLinks[leftBound];
1492                var elements = GetRightElements(leftLink, nextLeftLink);
1493                if (leftBound <= rightBound)
1494                {
1495                    for (var i = elements.Length - 1; i >= 0; i--)
1496                    {
1497                        var element = elements[i];
1498                        if (element != 0)
1499                        {
1500                            CollectMatchingSequences(element, leftBound + 1, middleLinks,
                                ↪ rightLink, rightBound, ref results);
1501                        }
1502                    }
1503                }
1504                else
1505                {
1506                    for (var i = elements.Length - 1; i >= 0; i--)
1507                    {
1508                        var element = elements[i];
1509                        if (element != 0)
1510                        {
1511                            results.Add(element);
1512                        }
1513                    }
1514                }
1515            }
1516            else
1517            {
1518                var nextRightLink = middleLinks[rightBound];
1519                var elements = GetLeftElements(rightLink, nextRightLink);
1520                if (leftBound <= rightBound)
1521                {
1522                    for (var i = elements.Length - 1; i >= 0; i--)
1523                    {
1524                        var element = elements[i];
1525                        if (element != 0)
1526                        {
1527                            CollectMatchingSequences(leftLink, leftBound, middleLinks,
                                ↪ elements[i], rightBound - 1, ref results);
1528                        }
1529                    }
1530                }
1531                else
1532                {
1533                    for (var i = elements.Length - 1; i >= 0; i--)
1534                    {
1535                        var element = elements[i];
1536                        if (element != 0)
1537                        {
1538                            results.Add(element);
1539                        }
1540                    }
1541                }
1542            }
1543        }
1544
1545        public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1546        {
1547            var result = new ulong[5];
1548            TryStepRight(startLink, rightLink, result, 0);
1549            Links.Each(Constants.Any, startLink, couple =>
1550            {
1551                if (couple != startLink)
1552                {
1553                    if (TryStepRight(couple, rightLink, result, 2))
1554                    {
1555                        return false;
1556                    }
```

```
                }
                return true;
            });
            if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
            {
                result[4] = startLink;
            }
            return result;
        }

        public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
        {
            var added = 0;
            Links.Each(startLink, Constants.Any, couple =>
            {
                if (couple != startLink)
                {
                    var coupleTarget = Links.GetTarget(couple);
                    if (coupleTarget == rightLink)
                    {
                        result[offset] = couple;
                        if (++added == 2)
                        {
                            return false;
                        }
                    }
                    else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
                    ↪  == Net.And &&
                    {
                        result[offset + 1] = couple;
                        if (++added == 2)
                        {
                            return false;
                        }
                    }
                }
                return true;
            });
            return added > 0;
        }

        public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
        {
            var result = new ulong[5];
            TryStepLeft(startLink, leftLink, result, 0);
            Links.Each(startLink, Constants.Any, couple =>
            {
                if (couple != startLink)
                {
                    if (TryStepLeft(couple, leftLink, result, 2))
                    {
                        return false;
                    }
                }
                return true;
            });
            if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
            {
                result[4] = leftLink;
            }
            return result;
        }

        public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
        {
            var added = 0;
            Links.Each(Constants.Any, startLink, couple =>
            {
                if (couple != startLink)
                {
                    var coupleSource = Links.GetSource(couple);
                    if (coupleSource == leftLink)
                    {
                        result[offset] = couple;
                        if (++added == 2)
                        {
                            return false;
                        }
                    }
```

```
                    else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
                    ↪  == Net.And &&
                    {
                        result[offset + 1] = couple;
                        if (++added == 2)
                        {
                            return false;
                        }
                    }
                }
                return true;
            });
            return added > 0;
        }

        #endregion

        #region Walkers

        public class PatternMatcher : RightSequenceWalker<ulong>
        {
            private readonly Sequences _sequences;
            private readonly ulong[] _patternSequence;
            private readonly HashSet<LinkIndex> _linksInSequence;
            private readonly HashSet<LinkIndex> _results;

            #region Pattern Match

            enum PatternBlockType
            {
                Undefined,
                Gap,
                Elements
            }

            struct PatternBlock
            {
                public PatternBlockType Type;
                public long Start;
                public long Stop;
            }

            private readonly List<PatternBlock> _pattern;
            private int _patternPosition;
            private long _sequencePosition;

            #endregion

            public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
            ↪  HashSet<LinkIndex> results)
                : base(sequences.Links.Unsync, new DefaultStack<ulong>())
            {
                _sequences = sequences;
                _patternSequence = patternSequence;
                _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                ↪  _sequences.Constants.Any && x != ZeroOrMany));
                _results = results;
                _pattern = CreateDetailedPattern();
            }

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

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

            private List<PatternBlock> CreateDetailedPattern()
            {
                var pattern = new List<PatternBlock>();
```

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

```
1791                    }
1792                    return pattern;
1793                }
1794
1795                // match: search for regexp anywhere in text
1796                //int match(char* regexp, char* text)
1797                //{
1798                //    do
1799                //    {
1800                //    } while (*text++ != '\0');
1801                //    return 0;
1802                //}
1803
1804                // matchhere: search for regexp at beginning of text
1805                //int matchhere(char* regexp, char* text)
1806                //{
1807                //    if (regexp[0] == '\0')
1808                //        return 1;
1809                //    if (regexp[1] == '*')
1810                //        return matchstar(regexp[0], regexp + 2, text);
1811                //    if (regexp[0] == '$' && regexp[1] == '\0')
1812                //        return *text == '\0';
1813                //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1814                //        return matchhere(regexp + 1, text + 1);
1815                //    return 0;
1816                //}
1817
1818                // matchstar: search for c*regexp at beginning of text
1819                //int matchstar(int c, char* regexp, char* text)
1820                //{
1821                //    do
1822                //    {    /* a * matches zero or more instances */
1823                //        if (matchhere(regexp, text))
1824                //            return 1;
1825                //    } while (*text != '\0' && (*text++ == c || c == '.'));
1826                //    return 0;
1827                //}
1828
1829                //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
                    ↪  long maximumGap)
1830                //{
1831                //    mininumGap = 0;
1832                //    maximumGap = 0;
1833                //    element = 0;
1834                //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1835                //    {
1836                //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1837                //            mininumGap++;
1838                //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1839                //            maximumGap = long.MaxValue;
1840                //        else
1841                //            break;
1842                //    }
1843
1844                //    if (maximumGap < mininumGap)
1845                //        maximumGap = mininumGap;
1846                //}
1847
1848                private bool PatternMatchCore(LinkIndex element)
1849                {
1850                    if (_patternPosition >= _pattern.Count)
1851                    {
1852                        _patternPosition = -2;
1853                        return false;
1854                    }
1855                    var currentPatternBlock = _pattern[_patternPosition];
1856                    if (currentPatternBlock.Type == PatternBlockType.Gap)
1857                    {
1858                        //var currentMatchingBlockLength = (_sequencePosition -
                            ↪  _lastMatchedBlockPosition);
1859                        if (_sequencePosition < currentPatternBlock.Start)
1860                        {
1861                            _sequencePosition++;
1862                            return true; // Двигаемся дальше
1863                        }
1864                        // Это последний блок
1865                        if (_pattern.Count == _patternPosition + 1)
1866                        {
```

```
                    _patternPosition++;
                    _sequencePosition = 0;
                    return false; // Полное соответствие
                }
                else
                {
                    if (_sequencePosition > currentPatternBlock.Stop)
                    {
                        return false; // Соответствие невозможно
                    }
                    var nextPatternBlock = _pattern[_patternPosition + 1];
                    if (_patternSequence[nextPatternBlock.Start] == element)
                    {
                        if (nextPatternBlock.Start < nextPatternBlock.Stop)
                        {
                            _patternPosition++;
                            _sequencePosition = 1;
                        }
                        else
                        {
                            _patternPosition += 2;
                            _sequencePosition = 0;
                        }
                    }
                }
            }
            else // currentPatternBlock.Type == PatternBlockType.Elements
            {
                var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
                if (_patternSequence[patternElementPosition] != element)
                {
                    return false; // Соответствие невозможно
                }
                if (patternElementPosition == currentPatternBlock.Stop)
                {
                    _patternPosition++;
                    _sequencePosition = 0;
                }
                else
                {
                    _sequencePosition++;
                }
            }
            return true;
            //if (_patternSequence[_patternPosition] != element)
            //    return false;
            //else
            //{
            //    _sequencePosition++;
            //    _patternPosition++;
            //    return true;
            //}
            ////////
            //if (_filterPosition == _patternSequence.Length)
            //{
            //    _filterPosition = -2; // Длиннее чем нужно
            //    return false;
            //}
            //if (element != _patternSequence[_filterPosition])
            //{
            //    _filterPosition = -1;
            //    return false; // Начинается иначе
            //}
            //_filterPosition++;
            //if (_filterPosition == (_patternSequence.Length - 1))
            //    return false;
            //if (_filterPosition >= 0)
            //{
            //    if (element == _patternSequence[_filterPosition + 1])
            //        _filterPosition++;
            //    else
            //        return false;
            //}
            //if (_filterPosition < 0)
            //{
            //    if (element == _patternSequence[0])
            //        _filterPosition = 0;
            //}
        }
```

```
1946
1947                public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
1948                {
1949                    foreach (var sequenceToMatch in sequencesToMatch)
1950                    {
1951                        if (PatternMatch(sequenceToMatch))
1952                        {
1953                            _results.Add(sequenceToMatch);
1954                        }
1955                    }
1956                }
1957            }
1958
1959            #endregion
1960        }
1961    }
```

1.82 ./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          public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
62          {
63              Links = links;
64              _sync = links.SyncRoot;
65              Options = options;
66              Options.ValidateOptions();
67              Options.InitOptions(Links);
68              Constants = links.Constants;
69          }
70
71          public Sequences(SynchronizedLinks<LinkIndex> links)
72              : this(links, new SequencesOptions<LinkIndex>())
73          {
74          }
75
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          private LinkIndex GetSequenceElements(LinkIndex sequence)
99          {
100             if (Options.UseSequenceMarker)
101             {
102                 var linkContents = new Link<ulong>(Links.GetLink(sequence));
103                 if (linkContents.Source == Options.SequenceMarkerLink)
104                 {
105                     return linkContents.Target;
106                 }
107                 if (linkContents.Target == Options.SequenceMarkerLink)
108                 {
109                     return linkContents.Source;
110                 }
111             }
112             return sequence;
113         }
114
115         #region Count
116
117         public LinkIndex Count(IList<LinkIndex> restrictions)
118         {
119             if (restrictions.IsNullOrEmpty())
120             {
121                 return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
122             }
123             if (restrictions.Count == 1) // Первая связь это адрес
124             {
125                 var sequenceIndex = restrictions[0];
126                 if (sequenceIndex == Constants.Null)
127                 {
128                     return 0;
129                 }
130                 if (sequenceIndex == Constants.Any)
131                 {
132                     return Count(null);
133                 }
```

```
                if (Options.UseSequenceMarker)
                {
                    return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
                }
                return Links.Exists(sequenceIndex) ? 1UL : 0;
            }
            throw new NotImplementedException();
        }

        private LinkIndex CountUsages(params LinkIndex[] restrictions)
        {
            if (restrictions.Length == 0)
            {
                return 0;
            }
            if (restrictions.Length == 1) // Первая связь это адрес
            {
                if (restrictions[0] == Constants.Null)
                {
                    return 0;
                }
                var any = Constants.Any;
                if (Options.UseSequenceMarker)
                {
                    var elementsLink = GetSequenceElements(restrictions[0]);
                    var sequenceLink = GetSequenceByElements(elementsLink);
                    if (sequenceLink != Constants.Null)
                    {
                        return Links.Count(any, sequenceLink) + Links.Count(any, elementsLink) -
                        ↪  1;
                    }
                    return Links.Count(any, elementsLink);
                }
                return Links.Count(any, restrictions[0]);
            }
            throw new NotImplementedException();
        }

        #endregion

        #region Create

        public LinkIndex Create(IList<LinkIndex> restrictions)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (restrictions.IsNullOrEmpty())
                {
                    return Constants.Null;
                }
                Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
                return CreateCore(restrictions);
            });
        }

        private LinkIndex CreateCore(IList<LinkIndex> restrictions)
        {
            LinkIndex[] sequence = restrictions.SkipFirst();
            if (Options.UseIndex)
            {
                Options.Index.Add(sequence);
            }
            var sequenceRoot = default(LinkIndex);
            if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
            {
                var matches = Each(restrictions);
                if (matches.Count > 0)
                {
                    sequenceRoot = matches[0];
                }
            }
            else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
            {
                return CompactCore(sequence);
            }
            if (sequenceRoot == default)
            {
                sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
```

```
211             }
212             if (Options.UseSequenceMarker)
213             {
214                 return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
215             }
216             return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
217         }
218 
219         #endregion
220 
221         #region Each
222 
223         public List<LinkIndex> Each(IList<LinkIndex> sequence)
224         {
225             var results = new List<LinkIndex>();
226             var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
227             Each(filler.AddFirstAndReturnConstant, sequence);
228             return results;
229         }
230 
231         public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
        ↪  restrictions)
232         {
233             return _sync.ExecuteReadOperation(() =>
234             {
235                 if (restrictions.IsNullOrEmpty())
236                 {
237                     return Constants.Continue;
238                 }
239                 Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
240                 if (restrictions.Count == 1)
241                 {
242                     var link = restrictions[0];
243                     var any = Constants.Any;
244                     if (link == any)
245                     {
246                         if (Options.UseSequenceMarker)
247                         {
248                             return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
                            ↪  Options.SequenceMarkerLink, any));
249                         }
250                         else
251                         {
252                             return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
                            ↪  any));
253                         }
254                     }
255                     if (Options.UseSequenceMarker)
256                     {
257                         var sequenceLinkValues = Links.Unsync.GetLink(link);
258                         if (sequenceLinkValues[Constants.SourcePart] ==
                        ↪  Options.SequenceMarkerLink)
259                         {
260                             link = sequenceLinkValues[Constants.TargetPart];
261                         }
262                     }
263                     var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
264                     sequence[0] = link;
265                     return handler(sequence);
266                 }
267                 else if (restrictions.Count == 2)
268                 {
269                     throw new NotImplementedException();
270                 }
271                 else if (restrictions.Count == 3)
272                 {
273                     return Links.Unsync.Each(handler, restrictions);
274                 }
275                 else
276                 {
277                     var sequence = restrictions.SkipFirst();
278                     if (Options.UseIndex && !Options.Index.MightContain(sequence))
279                     {
280                         return Constants.Break;
281                     }
282                     return EachCore(handler, sequence);
283                 }
284             });
```

```
285         }
286
287         private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
            ↪ values)
288         {
289             var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
290             // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
                ↪ Id.
291             Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
                ↪ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
                ↪ matcher.HandleFullMatched;
292             //if (sequence.Length >= 2)
293             if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
294             {
295                 return Constants.Break;
296             }
297             var last = values.Count - 2;
298             for (var i = 1; i < last; i++)
299             {
300                 if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
                    ↪ Constants.Continue)
301                 {
302                     return Constants.Break;
303                 }
304             }
305             if (values.Count >= 3)
306             {
307                 if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
                    ↪ != Constants.Continue)
308                 {
309                     return Constants.Break;
310                 }
311             }
312             return Constants.Continue;
313         }
314
315         private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
            ↪ left, LinkIndex right)
316         {
317             return Links.Unsync.Each(doublet =>
318             {
319                 var doubletIndex = doublet[Constants.IndexPart];
320                 if (StepRight(handler, doubletIndex, right) != Constants.Continue)
321                 {
322                     return Constants.Break;
323                 }
324                 if (left != doubletIndex)
325                 {
326                     return PartialStepRight(handler, doubletIndex, right);
327                 }
328                 return Constants.Continue;
329             }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
330         }
331
332         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));
333
334         private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
            ↪ right, LinkIndex stepFrom)
335         {
336             var upStep = stepFrom;
337             var firstSource = Links.Unsync.GetTarget(upStep);
338             while (firstSource != right && firstSource != upStep)
339             {
340                 upStep = firstSource;
341                 firstSource = Links.Unsync.GetSource(upStep);
342             }
343             if (firstSource == right)
344             {
345                 return handler(new LinkAddress<LinkIndex>(stepFrom));
346             }
347             return Constants.Continue;
348         }
349
```

```
350         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));
351
352         private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
            ↪ left, LinkIndex 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                 return handler(new LinkAddress<LinkIndex>(stepFrom));
364             }
365             return Constants.Continue;
366         }
367
368         #endregion
369
370         #region Update
371
372         public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
373         {
374             var sequence = restrictions.SkipFirst();
375             var newSequence = substitution.SkipFirst();
376
377             if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
378             {
379                 return Constants.Null;
380             }
381             if (sequence.IsNullOrEmpty())
382             {
383                 return Create(substitution);
384             }
385             if (newSequence.IsNullOrEmpty())
386             {
387                 Delete(restrictions);
388                 return Constants.Null;
389             }
390             return _sync.ExecuteWriteOperation((Func<ulong>)(() =>
391             {
392                 ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
393                 Links.EnsureLinkExists(newSequence);
394                 return UpdateCore(sequence, newSequence);
395             }));
396         }
397
398         private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence)
399         {
400             LinkIndex bestVariant;
401             if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
                ↪ !sequence.EqualTo(newSequence))
402             {
403                 bestVariant = CompactCore(newSequence);
404             }
405             else
406             {
407                 bestVariant = CreateCore(newSequence);
408             }
409             // TODO: Check all options only ones before loop execution
410             // Возможно нужно две версии Each, возвращающий фактические последовательности и с
                ↪ маркером,
411             // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
                ↪ можно получить имея только фактические последовательности.
412             foreach (var variant in Each(sequence))
413             {
414                 if (variant != bestVariant)
415                 {
416                     UpdateOneCore(variant, bestVariant);
417                 }
418             }
419             return bestVariant;
420         }
421
```

```
422         private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
423         {
424             if (Options.UseGarbageCollection)
425             {
426                 var sequenceElements = GetSequenceElements(sequence);
427                 var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
428                 var sequenceLink = GetSequenceByElements(sequenceElements);
429                 var newSequenceElements = GetSequenceElements(newSequence);
430                 var newSequenceLink = GetSequenceByElements(newSequenceElements);
431                 if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
432                 {
433                     if (sequenceLink != Constants.Null)
434                     {
435                         Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
436                     }
437                     Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
438                 }
439                 ClearGarbage(sequenceElementsContents.Source);
440                 ClearGarbage(sequenceElementsContents.Target);
441             }
442             else
443             {
444                 if (Options.UseSequenceMarker)
445                 {
446                     var sequenceElements = GetSequenceElements(sequence);
447                     var sequenceLink = GetSequenceByElements(sequenceElements);
448                     var newSequenceElements = GetSequenceElements(newSequence);
449                     var newSequenceLink = GetSequenceByElements(newSequenceElements);
450                     if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
451                     {
452                         if (sequenceLink != Constants.Null)
453                         {
454                             Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
455                         }
456                         Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
457                     }
458                 }
459                 else
460                 {
461                     if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
462                     {
463                         Links.Unsync.MergeAndDelete(sequence, newSequence);
464                     }
465                 }
466             }
467         }
468 
469         #endregion
470 
471         #region Delete
472 
473         public void Delete(IList<LinkIndex> restrictions)
474         {
475             _sync.ExecuteWriteOperation(() =>
476             {
477                 var sequence = restrictions.SkipFirst();
478                 // TODO: Check all options only ones before loop execution
479                 foreach (var linkToDelete in Each(sequence))
480                 {
481                     DeleteOneCore(linkToDelete);
482                 }
483             });
484         }
485 
486         private void DeleteOneCore(LinkIndex link)
487         {
488             if (Options.UseGarbageCollection)
489             {
490                 var sequenceElements = GetSequenceElements(link);
491                 var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
492                 var sequenceLink = GetSequenceByElements(sequenceElements);
493                 if (Options.UseCascadeDelete || CountUsages(link) == 0)
494                 {
495                     if (sequenceLink != Constants.Null)
496                     {
497                         Links.Unsync.Delete(sequenceLink);
498                     }
499                     Links.Unsync.Delete(link);
```

```
500                 }
501                 ClearGarbage(sequenceElementsContents.Source);
502                 ClearGarbage(sequenceElementsContents.Target);
503             }
504             else
505             {
506                 if (Options.UseSequenceMarker)
507                 {
508                     var sequenceElements = GetSequenceElements(link);
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                 }
519                 else
520                 {
521                     if (Options.UseCascadeDelete || CountUsages(link) == 0)
522                     {
523                         Links.Unsync.Delete(link);
524                     }
525                 }
526             }
527         }
528 
529         #endregion
530 
531         #region Compactification
532 
533         public void CompactAll()
534         {
535             _sync.ExecuteWriteOperation(() =>
536             {
537                 var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
538                 for (int i = 0; i < sequences.Count; i++)
539                 {
540                     var sequence = this.ToList(sequences[i]);
541                     Compact(sequence.ShiftRight());
542                 }
543             });
544         }
545 
546         /// <remarks>
547         /// bestVariant можно выбирать по максимальному числу использований,
548         /// но балансированный позволяет гарантировать уникальность (если есть возможность,
549         /// гарантировать его использование в других местах).
550         ///
551         /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
552         /// </remarks>
553         public LinkIndex Compact(IList<LinkIndex> sequence)
554         {
555             return _sync.ExecuteWriteOperation(() =>
556             {
557                 if (sequence.IsNullOrEmpty())
558                 {
559                     return Constants.Null;
560                 }
561                 Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
562                 return CompactCore(sequence);
563             });
564         }
565 
566         [MethodImpl(MethodImplOptions.AggressiveInlining)]
567         private LinkIndex CompactCore(IList<LinkIndex> sequence) => UpdateCore(sequence,
            ↪ sequence);
568 
569         #endregion
570 
571         #region Garbage Collection
572 
573         /// <remarks>
574         /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
            ↪ определить извне или в унаследованном классе
575         /// </remarks>
576         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
        ↪  !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;

        private void ClearGarbage(LinkIndex link)
        {
            if (IsGarbage(link))
            {
                var contents = new Link<ulong>(Links.GetLink(link));
                Links.Unsync.Delete(link);
                ClearGarbage(contents.Source);
                ClearGarbage(contents.Target);
            }
        }

        #endregion

        #region Walkers

        public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var links = Links.Unsync;
                foreach (var part in Options.Walker.Walk(sequence))
                {
                    if (!handler(part))
                    {
                        return false;
                    }
                }
                return true;
            });
        }

        public class Matcher : RightSequenceWalker<LinkIndex>
        {
            private readonly Sequences _sequences;
            private readonly IList<LinkIndex> _patternSequence;
            private readonly HashSet<LinkIndex> _linksInSequence;
            private readonly HashSet<LinkIndex> _results;
            private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
            private readonly HashSet<LinkIndex> _readAsElements;
            private int _filterPosition;

            public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
            ↪  HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
            ↪  HashSet<LinkIndex> readAsElements = null)
                : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
            {
                _sequences = sequences;
                _patternSequence = patternSequence;
                _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                ↪  Links.Constants.Any && x != ZeroOrMany));
                _results = results;
                _stopableHandler = stopableHandler;
                _readAsElements = readAsElements;
            }

            protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
            ↪  (_readAsElements != null && _readAsElements.Contains(link)) ||
            ↪  _linksInSequence.Contains(link);

            public bool FullMatch(LinkIndex sequenceToMatch)
            {
                _filterPosition = 0;
                foreach (var part in Walk(sequenceToMatch))
                {
                    if (!FullMatchCore(part))
                    {
                        break;
                    }
                }
                return _filterPosition == _patternSequence.Count;
            }

            private bool FullMatchCore(LinkIndex element)
            {
                if (_filterPosition == _patternSequence.Count)
                {
                    _filterPosition = -2; // Длиннее чем нужно
```

```
                return false;
            }
            if (_patternSequence[_filterPosition] != Links.Constants.Any
             && element != _patternSequence[_filterPosition])
            {
                _filterPosition = -1;
                return false; // Начинается/Продолжается иначе
            }
            _filterPosition++;
            return true;
        }

        public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
        {
            var sequenceToMatch = restrictions[Links.Constants.IndexPart];
            if (FullMatch(sequenceToMatch))
            {
                _results.Add(sequenceToMatch);
            }
        }

        public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
        {
            var sequenceToMatch = restrictions[Links.Constants.IndexPart];
            if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
            {
                return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
            }
            return Links.Constants.Continue;
        }

        public LinkIndex HandleFullMatchedSequence(IList<LinkIndex> restrictions)
        {
            var sequenceToMatch = restrictions[Links.Constants.IndexPart];
            var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
            if (sequence != Links.Constants.Null && FullMatch(sequenceToMatch) &&
            ↪  _results.Add(sequenceToMatch))
            {
                return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
            }
            return Links.Constants.Continue;
        }

        /// <remarks>
        /// TODO: Add support for LinksConstants.Any
        /// </remarks>
        public bool PartialMatch(LinkIndex sequenceToMatch)
        {
            _filterPosition = -1;
            foreach (var part in Walk(sequenceToMatch))
            {
                if (!PartialMatchCore(part))
                {
                    break;
                }
            }
            return _filterPosition == _patternSequence.Count - 1;
        }

        private bool PartialMatchCore(LinkIndex element)
        {
            if (_filterPosition == (_patternSequence.Count - 1))
            {
                return false; // Нашлось
            }
            if (_filterPosition >= 0)
            {
                if (element == _patternSequence[_filterPosition + 1])
                {
                    _filterPosition++;
                }
                else
                {
                    _filterPosition = -1;
                }
            }
            if (_filterPosition < 0)
            {
                if (element == _patternSequence[0])
```

```
                    {
                        _filterPosition = 0;
                    }
                }
                return true; // Ищем дальше
            }

            public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
            {
                if (PartialMatch(sequenceToMatch))
                {
                    _results.Add(sequenceToMatch);
                }
            }

            public LinkIndex HandlePartialMatched(IList<LinkIndex> restrictions)
            {
                var sequenceToMatch = restrictions[Links.Constants.IndexPart];
                if (PartialMatch(sequenceToMatch))
                {
                    return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
                }
                return Links.Constants.Continue;
            }

            public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)
            {
                foreach (var sequenceToMatch in sequencesToMatch)
                {
                    if (PartialMatch(sequenceToMatch))
                    {
                        _results.Add(sequenceToMatch);
                    }
                }
            }

            public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
            ↪  sequencesToMatch)
            {
                foreach (var sequenceToMatch in sequencesToMatch)
                {
                    if (PartialMatch(sequenceToMatch))
                    {
                        _readAsElements.Add(sequenceToMatch);
                        _results.Add(sequenceToMatch);
                    }
                }
            }
        }

        #endregion
    }
}
```

1.83 ./Platform.Data.Doublets/Sequences/SequencesExtensions.cs

```
using System;
using System.Collections.Generic;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences
{
    public static class SequencesExtensions
    {
        public static TLink Create<TLink>(this ILinks<TLink> sequences, IList<TLink[]>
        ↪  groupedSequence)
        {
            var finalSequence = new TLink[groupedSequence.Count];
            for (var i = 0; i < finalSequence.Length; i++)
            {
                var part = groupedSequence[i];
                finalSequence[i] = part.Length == 1 ? part[0] :
                ↪  sequences.Create(part.ShiftRight());
            }
            return sequences.Create(finalSequence.ShiftRight());
        }

        public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
```

```
23          {
24              var list = new List<TLink>();
25              var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
26              sequences.Each(filler.AddSkipFirstAndReturnConstant, new
                ↪ LinkAddress<TLink>(sequence));
27              return list;
28          }
29      }
30  }
```

1.84 ./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
13  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14
15  namespace Platform.Data.Doublets.Sequences
16  {
17      public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
        ↪ ILinks<TLink> must contain GetConstants function.
18      {
19          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
20
21          public TLink SequenceMarkerLink { get; set; }
22          public bool UseCascadeUpdate { get; set; }
23          public bool UseCascadeDelete { get; set; }
24          public bool UseIndex { get; set; } // TODO: Update Index on sequence update/delete.
25          public bool UseSequenceMarker { get; set; }
26          public bool UseCompression { get; set; }
27          public bool UseGarbageCollection { get; set; }
28          public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get; set; }
29          public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }
30
31          public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher { get; set; }
32          public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
33          public ISequenceIndex<TLink> Index { get; set; }
34          public ISequenceWalker<TLink> Walker { get; set; }
35          public bool ReadFullSequence { get; set; }
36
37          // TODO: Реализовать компактификацию при чтении
38          //public bool EnforceSingleSequenceVersionOnRead { get; set; }
39          //public bool UseRequestMarker { get; set; }
40          //public bool StoreRequestResults { get; set; }
41
42          public void InitOptions(ISynchronizedLinks<TLink> links)
43          {
44              if (UseSequenceMarker)
45              {
46                  if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
47                  {
48                      SequenceMarkerLink = links.CreatePoint();
49                  }
50                  else
51                  {
52                      if (!links.Exists(SequenceMarkerLink))
53                      {
54                          var link = links.CreatePoint();
55                          if (!_equalityComparer.Equals(link, SequenceMarkerLink))
56                          {
57                              throw new InvalidOperationException("Cannot recreate sequence marker
                                ↪ link.");
58                          }
59                      }
60                  }
61                  if (MarkedSequenceMatcher == null)
62                  {
63                      MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
                        ↪ SequenceMarkerLink);
64                  }
```

```
            }
            var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
            if (UseCompression)
            {
                if (LinksToSequenceConverter == null)
                {
                    ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
                    if (UseSequenceMarker)
                    {
                        totalSequenceSymbolFrequencyCounter = new
                        ↪  TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
                        ↪  MarkedSequenceMatcher);
                    }
                    else
                    {
                        totalSequenceSymbolFrequencyCounter = new
                        ↪  TotalSequenceSymbolFrequencyCounter<TLink>(links);
                    }
                    var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
                    ↪  totalSequenceSymbolFrequencyCounter);
                    var compressingConverter = new CompressingConverter<TLink>(links,
                    ↪  balancedVariantConverter, doubletFrequenciesCache);
                    LinksToSequenceConverter = compressingConverter;
                }
            }
            else
            {
                if (LinksToSequenceConverter == null)
                {
                    LinksToSequenceConverter = balancedVariantConverter;
                }
            }
            if (UseIndex && Index == null)
            {
                Index = new SequenceIndex<TLink>(links);
            }
            if (Walker == null)
            {
                Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
            }
        }

        public void ValidateOptions()
        {
            if (UseGarbageCollection && !UseSequenceMarker)
            {
                throw new NotSupportedException("To use garbage collection UseSequenceMarker
                ↪  option must be on.");
            }
        }
    }
}
```

1.85 ./Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs

```
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public interface ISequenceWalker<TLink>
    {
        IEnumerable<TLink> Walk(TLink sequence);
    }
}
```

1.86 ./Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Stacks;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
    {
```

```
        public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪  isElement) : base(links, stack, isElement) { }

        public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
        ↪  links.IsPartialPoint) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPop(TLink element) =>
        ↪  Links.GetSource(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPush(TLink element) =>
        ↪  Links.GetTarget(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override IEnumerable<TLink> WalkContents(TLink element)
        {
            var parts = Links.GetLink(element);
            var start = Links.Constants.IndexPart + 1;
            for (var i = parts.Count - 1; i >= start; i--)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
    }
}
```

1.87 ./Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

//#define USEARRAYPOOL
#if USEARRAYPOOL
using Platform.Collections;
#endif

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly Func<TLink, bool> _isElement;

        public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
        ↪  base(links) => _isElement = isElement;

        public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
        ↪  Links.IsPartialPoint;

        public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);

        public TLink[] ToArray(TLink sequence)
        {
            var length = 1;
            var array = new TLink[length];
            array[0] = sequence;
            if (_isElement(sequence))
            {
                return array;
            }
            bool hasElements;
            do
            {
                length *= 2;
#if USEARRAYPOOL
                var nextArray = ArrayPool.Allocate<ulong>(length);
#else
                var nextArray = new TLink[length];
#endif
                hasElements = false;
                for (var i = 0; i < array.Length; i++)
```

```
                {
                    var candidate = array[i];
                    if (_equalityComparer.Equals(array[i], default))
                    {
                        continue;
                    }
                    var doubletOffset = i * 2;
                    if (_isElement(candidate))
                    {
                        nextArray[doubletOffset] = candidate;
                    }
                    else
                    {
                        var link = Links.GetLink(candidate);
                        var linkSource = Links.GetSource(link);
                        var linkTarget = Links.GetTarget(link);
                        nextArray[doubletOffset] = linkSource;
                        nextArray[doubletOffset + 1] = linkTarget;
                        if (!hasElements)
                        {
                            hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
                        }
                    }
                }
#if USEARRAYPOOL
                if (array.Length > 1)
                {
                    ArrayPool.Free(array);
                }
#endif
                array = nextArray;
            }
            while (hasElements);
            var filledElementsCount = CountFilledElements(array);
            if (filledElementsCount == array.Length)
            {
                return array;
            }
            else
            {
                return CopyFilledElements(array, filledElementsCount);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
        {
            var finalArray = new TLink[filledElementsCount];
            for (int i = 0, j = 0; i < array.Length; i++)
            {
                if (!_equalityComparer.Equals(array[i], default))
                {
                    finalArray[j] = array[i];
                    j++;
                }
            }
#if USEARRAYPOOL
                ArrayPool.Free(array);
#endif
            return finalArray;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static int CountFilledElements(TLink[] array)
        {
            var count = 0;
            for (var i = 0; i < array.Length; i++)
            {
                if (!_equalityComparer.Equals(array[i], default))
                {
                    count++;
                }
            }
            return count;
        }
    }
}
```

1.88 ./Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Stacks;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
    {
        public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪  isElement) : base(links, stack, isElement) { }

        public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
        ↪  stack, links.IsPartialPoint) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPop(TLink element) =>
        ↪  Links.GetTarget(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPush(TLink element) =>
        ↪  Links.GetSource(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override IEnumerable<TLink> WalkContents(TLink element)
        {
            var parts = Links.GetLink(element);
            for (var i = Links.Constants.IndexPart + 1; i < parts.Count; i++)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
    }
}
```

1.89 ./Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Stacks;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Sequences.Walkers
{
    public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
    ↪  ISequenceWalker<TLink>
    {
        private readonly IStack<TLink> _stack;
        private readonly Func<TLink, bool> _isElement;

        protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪  isElement) : base(links)
        {
            _stack = stack;
            _isElement = isElement;
        }

        protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
        ↪  stack, links.IsPartialPoint)
        {
        }

        public IEnumerable<TLink> Walk(TLink sequence)
        {
            _stack.Clear();
            var element = sequence;
            if (IsElement(element))
            {
                yield return element;
            }
            else
            {
```

```
                while (true)
                {
                    if (IsElement(element))
                    {
                        if (_stack.IsEmpty)
                        {
                            break;
                        }
                        element = _stack.Pop();
                        foreach (var output in WalkContents(element))
                        {
                            yield return output;
                        }
                        element = GetNextElementAfterPop(element);
                    }
                    else
                    {
                        _stack.Push(element);
                        element = GetNextElementAfterPush(element);
                    }
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLink GetNextElementAfterPop(TLink element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLink GetNextElementAfterPush(TLink element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract IEnumerable<TLink> WalkContents(TLink element);
    }
}
```

1.90 ./Platform.Data.Doublets/Stacks/Stack.cs

```
using System.Collections.Generic;
using Platform.Collections.Stacks;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Stacks
{
    public class Stack<TLink> : IStack<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly ILinks<TLink> _links;
        private readonly TLink _stack;

        public bool IsEmpty => _equalityComparer.Equals(Peek(), _stack);

        public Stack(ILinks<TLink> links, TLink stack)
        {
            _links = links;
            _stack = stack;
        }

        private TLink GetStackMarker() => _links.GetSource(_stack);

        private TLink GetTop() => _links.GetTarget(_stack);

        public TLink Peek() => _links.GetTarget(GetTop());

        public TLink Pop()
        {
            var element = Peek();
            if (!_equalityComparer.Equals(element, _stack))
            {
                var top = GetTop();
                var previousTop = _links.GetSource(top);
                _links.Update(_stack, GetStackMarker(), previousTop);
                _links.Delete(top);
            }
            return element;
        }
```

```
41
42          public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
            ↪  _links.GetOrCreate(GetTop(), element));
43      }
44  }
```

1.91 ./Platform.Data.Doublets/Stacks/StackExtensions.cs

```
1   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3   namespace Platform.Data.Doublets.Stacks
4   {
5       public static class StackExtensions
6       {
7           public static TLink CreateStack<TLink>(this ILinks<TLink> links, TLink stackMarker)
8           {
9               var stackPoint = links.CreatePoint();
10              var stack = links.Update(stackPoint, stackMarker, stackPoint);
11              return stack;
12          }
13      }
14  }
```

1.92 ./Platform.Data.Doublets/SynchronizedLinks.cs

```
1   using System;
2   using System.Collections.Generic;
3   using Platform.Data.Doublets;
4   using Platform.Threading.Synchronization;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets
9   {
10      /// <remarks>
11      /// TODO: Autogeneration of synchronized wrapper (decorator).
12      /// TODO: Try to unfold code of each method using IL generation for performance improvements.
13      /// TODO: Or even to unfold multiple layers of implementations.
14      /// </remarks>
15      public class SynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress>
16      {
17          public LinksConstants<TLinkAddress> Constants { get; }
18          public ISynchronization SyncRoot { get; }
19          public ILinks<TLinkAddress> Sync { get; }
20          public ILinks<TLinkAddress> Unsync { get; }
21
22          public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
            ↪  ReaderWriterLockSynchronization(), links) { }
23
24          public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
25          {
26              SyncRoot = synchronization;
27              Sync = this;
28              Unsync = links;
29              Constants = links.Constants;
30          }
31
32          public TLinkAddress Count(IList<TLinkAddress> restriction) =>
            ↪  SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
33          public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
            ↪  IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
            ↪  restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
34          public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
            ↪  SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
35          public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
            ↪  substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
            ↪  Unsync.Update);
36          public void Delete(IList<TLinkAddress> restrictions) =>
            ↪  SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
37
38          //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
            ↪  IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
39          //{
40          //    if (restriction != null && substitution != null &&
            ↪  !substitution.EqualTo(restriction))
41          //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
            ↪  substitution, substitutedHandler, Unsync.Trigger);
42
43          //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
            ↪  substitutedHandler, Unsync.Trigger);
```

```
        //}
    }
}
```

1.93 ./Platform.Data.Doublets/UInt64LinksExtensions.cs

```
using System;
using System.Text;
using System.Collections.Generic;
using Platform.Singletons;
using Platform.Data.Doublets.Unicode;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets
{
    public static class UInt64LinksExtensions
    {
        public static readonly LinksConstants<ulong> Constants =
        ↪ Default<LinksConstants<ulong>>.Instance;

        public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);

        public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
        {
            if (sequence == null)
            {
                return false;
            }
            var constants = links.Constants;
            for (var i = 0; i < sequence.Length; i++)
            {
                if (sequence[i] == constants.Any)
                {
                    return true;
                }
            }
            return false;
        }

        public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
        ↪ Func<Link<ulong>, bool> isElement, bool renderIndex = false, bool renderDebug =
        ↪ false)
        {
            var sb = new StringBuilder();
            var visited = new HashSet<ulong>();
            links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
            ↪ innerSb.Append(link.Index), renderIndex, renderDebug);
            return sb.ToString();
        }

        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)
        {
            var sb = new StringBuilder();
            var visited = new HashSet<ulong>();
            links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
            ↪ renderDebug);
            return sb.ToString();
        }

        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)
        {
            if (sb == null)
            {
                throw new ArgumentNullException(nameof(sb));
            }
            if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
            ↪ Constants.Itself)
            {
                return;
            }
            if (links.Exists(linkIndex))
            {
                if (visited.Add(linkIndex))
                {
```

```
                    sb.Append('(');
                    var link = new Link<ulong>(links.GetLink(linkIndex));
                    if (renderIndex)
                    {
                        sb.Append(link.Index);
                        sb.Append(':');
                    }
                    if (link.Source == link.Index)
                    {
                        sb.Append(link.Index);
                    }
                    else
                    {
                        var source = new Link<ulong>(links.GetLink(link.Source));
                        if (isElement(source))
                        {
                            appendElement(sb, source);
                        }
                        else
                        {
                            links.AppendStructure(sb, visited, source.Index, isElement,
                                ↪  appendElement, renderIndex);
                        }
                    }
                    sb.Append(' ');
                    if (link.Target == link.Index)
                    {
                        sb.Append(link.Index);
                    }
                    else
                    {
                        var target = new Link<ulong>(links.GetLink(link.Target));
                        if (isElement(target))
                        {
                            appendElement(sb, target);
                        }
                        else
                        {
                            links.AppendStructure(sb, visited, target.Index, isElement,
                                ↪  appendElement, renderIndex);
                        }
                    }
                    sb.Append(')');
                }
                else
                {
                    if (renderDebug)
                    {
                        sb.Append('*');
                    }
                    sb.Append(linkIndex);
                }
            }
            else
            {
                if (renderDebug)
                {
                    sb.Append('~');
                }
                sb.Append(linkIndex);
            }
        }
    }
}
```

1.94 ./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.IO;
using System.Runtime.CompilerServices;
using System.Threading;
using System.Threading.Tasks;
using Platform.Disposables;
using Platform.Timestamps;
using Platform.Unsafe;
using Platform.IO;
using Platform.Data.Doublets.Decorators;
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 &amp;amp; TransactionIdCombined;
54          ///         }
55          ///     }
56          ///
57          ///     public UniqueTimestamp Timestamp
58          ///     {
59          ///         get
60          ///         {
61          ///             return (UniqueTimestamp)mask &amp;amp; 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              public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
                ↪  transactionId, Link<ulong> before, Link<ulong> after)
95              {
96                  TransactionId = transactionId;
97                  Before = before;
98                  After = after;
99                  Timestamp = uniqueTimestampFactory.Create();
100             }
101
102             public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
                ↪  transactionId, Link<ulong> before)
103                 : this(uniqueTimestampFactory, transactionId, before, default)
104             {
105             }
106
107             public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId)
108                 : this(uniqueTimestampFactory, transactionId, default, default)
109             {
110             }
111
112             public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
                ↪  {After}";
113
114             public override bool Equals(object obj) => obj is Transition transition ?
                ↪  Equals(transition) : false;
115
116             public override int GetHashCode() => (TransactionId, Before, After,
                ↪  Timestamp).GetHashCode();
117
118             public bool Equals(Transition other) => TransactionId == other.TransactionId &&
                ↪  Before == other.Before && After == other.After && Timestamp == other.Timestamp;
119
120             public static bool operator ==(Transition left, Transition right) =>
                ↪  left.Equals(right);
121
122             public static bool operator !=(Transition left, Transition right) => !(left ==
                ↪  right);
123         }
124
125         /// <remarks>
126         /// Другие варианты реализации транзакций (атомарности):
127         ///     1. Разделение хранения значения связи ((Source Target) или (Source Linker
                ↪  Target)) и индексов.
128         ///     2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
                ↪  потребуется решить вопрос
129         ///        со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
                ↪  пересечениями идентификаторов.
130         ///
131         /// Где хранить промежуточный список транзакций?
132         ///
133         /// В оперативной памяти:
134         ///  Минусы:
135         ///     1. Может усложнить систему, если она будет функционировать самостоятельно,
136         ///     так как нужно отдельно выделять память под список трансформаций.
137         ///     2. Выделенной оперативной памяти может не хватить, в том случае,
138         ///     если транзакция использует слишком много трансформаций.
139         ///         -> Можно использовать жёсткий диск для слишком длинных транзакций.
140         ///         -> Максимальный размер списка трансформаций можно ограничить / задать
                ↪  константой.
141         ///     3. При подтверждении транзакции (Commit) все трансформации записываются разом
                ↪  создавая задержку.
142         ///
143         /// На жёстком диске:
144         ///  Минусы:
145         ///     1. Длительный отклик, на запись каждой трансформации.
146         ///     2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
147         ///         -> Это может решаться упаковкой/исключением дублирующих операций.
148         ///         -> Также это может решаться тем, что короткие транзакции вообще
149         ///            не будут записываться в случае отката.
150         ///     3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
                ↪  операции (трансформации)
151         ///        будут записаны в лог.
152         ///
153         /// </remarks>
154         public class Transaction : DisposableBase
155         {
```

```
156             private readonly Queue<Transition> _transitions;
157             private readonly UInt64LinksTransactionsLayer _layer;
158             public bool IsCommitted { get; private set; }
159             public bool IsReverted { get; private set; }
160 
161             public Transaction(UInt64LinksTransactionsLayer layer)
162             {
163                 _layer = layer;
164                 if (_layer._currentTransactionId != 0)
165                 {
166                     throw new NotSupportedException("Nested transactions not supported.");
167                 }
168                 IsCommitted = false;
169                 IsReverted = false;
170                 _transitions = new Queue<Transition>();
171                 SetCurrentTransaction(layer, this);
172             }
173 
174             public void Commit()
175             {
176                 EnsureTransactionAllowsWriteOperations(this);
177                 while (_transitions.Count > 0)
178                 {
179                     var transition = _transitions.Dequeue();
180                     _layer._transitions.Enqueue(transition);
181                 }
182                 _layer._lastCommitedTransactionId = _layer._currentTransactionId;
183                 IsCommitted = true;
184             }
185 
186             private void Revert()
187             {
188                 EnsureTransactionAllowsWriteOperations(this);
189                 var transitionsToRevert = new Transition[_transitions.Count];
190                 _transitions.CopyTo(transitionsToRevert, 0);
191                 for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
192                 {
193                     _layer.RevertTransition(transitionsToRevert[i]);
194                 }
195                 IsReverted = true;
196             }
197 
198             public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
                ↪ Transaction transaction)
199             {
200                 layer._currentTransactionId = layer._lastCommitedTransactionId + 1;
201                 layer._currentTransactionTransitions = transaction._transitions;
202                 layer._currentTransaction = transaction;
203             }
204 
205             public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
206             {
207                 if (transaction.IsReverted)
208                 {
209                     throw new InvalidOperationException("Transation is reverted.");
210                 }
211                 if (transaction.IsCommitted)
212                 {
213                     throw new InvalidOperationException("Transation is commited.");
214                 }
215             }
216 
217             protected override void Dispose(bool manual, bool wasDisposed)
218             {
219                 if (!wasDisposed && _layer != null && !_layer.IsDisposed)
220                 {
221                     if (!IsCommitted && !IsReverted)
222                     {
223                         Revert();
224                     }
225                     _layer.ResetCurrentTransation();
226                 }
227             }
228         }
229 
230         public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
231 
232         private readonly string _logAddress;
233         private readonly FileStream _log;
```

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

```
307
308         private void CommitTransition(Transition transition)
309         {
310             if (_currentTransaction != null)
311             {
312                 Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
313             }
314             var transitions = GetCurrentTransitions();
315             transitions.Enqueue(transition);
316         }
317
318         private void RevertTransition(Transition transition)
319         {
320             if (transition.After.IsNull()) // Revert Deletion with Creation
321             {
322                 Links.Create();
323             }
324             else if (transition.Before.IsNull()) // Revert Creation with Deletion
325             {
326                 Links.Delete(transition.After.Index);
327             }
328             else // Revert Update
329             {
330                 Links.Update(new[] { transition.After.Index, transition.Before.Source,
                    ↪  transition.Before.Target });
331             }
332         }
333
334         private void ResetCurrentTransation()
335         {
336             _currentTransactionId = 0;
337             _currentTransactionTransitions = null;
338             _currentTransaction = null;
339         }
340
341         private void PushTransitions()
342         {
343             if (_log == null || _transitions == null)
344             {
345                 return;
346             }
347             for (var i = 0; i < _transitions.Count; i++)
348             {
349                 var transition = _transitions.Dequeue();
350
351                 _log.Write(transition);
352                 _lastCommitedTransition = transition;
353             }
354         }
355
356         private void TransitionsPusher()
357         {
358             while (!IsDisposed && _transitionsPusher != null)
359             {
360                 Thread.Sleep(DefaultPushDelay);
361                 PushTransitions();
362             }
363         }
364
365         public Transaction BeginTransaction() => new Transaction(this);
366
367         private void DisposeTransitions()
368         {
369             try
370             {
371                 var pusher = _transitionsPusher;
372                 if (pusher != null)
373                 {
374                     _transitionsPusher = null;
375                     pusher.Wait();
376                 }
377                 if (_transitions != null)
378                 {
379                     PushTransitions();
380                 }
381                 _log.DisposeIfPossible();
382                 FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
383             }
384             catch (Exception ex)
```

```
            {
                ex.Ignore();
            }
        }

        #region DisposalBase

        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                DisposeTransitions();
            }
            base.Dispose(manual, wasDisposed);
        }

        #endregion
    }
}
```

1.95 ./Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs

```
using Platform.Converters;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Unicode
{
    public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<char, TLink>
    {
        private static readonly UncheckedConverter<char, TLink> _charToAddressConverter =
        ↪ UncheckedConverter<char, TLink>.Default;

        private readonly IConverter<TLink> _addressToNumberConverter;
        private readonly TLink _unicodeSymbolMarker;

        public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
        ↪ addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
        {
            _addressToNumberConverter = addressToNumberConverter;
            _unicodeSymbolMarker = unicodeSymbolMarker;
        }

        public TLink Convert(char source)
        {
            var unaryNumber =
            ↪ _addressToNumberConverter.Convert(_charToAddressConverter.Convert(source));
            return Links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
        }
    }
}
```

1.96 ./Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs

```
using System.Collections.Generic;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Indexes;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Unicode
{
    public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<string, TLink>
    {
        private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;
        private readonly ISequenceIndex<TLink> _index;
        private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
        private readonly TLink _unicodeSequenceMarker;

        public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
        ↪ charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
        ↪ TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
        {
            _charToUnicodeSymbolConverter = charToUnicodeSymbolConverter;
            _index = index;
            _listToSequenceLinkConverter = listToSequenceLinkConverter;
            _unicodeSequenceMarker = unicodeSequenceMarker;
        }

        public TLink Convert(string source)
```

```
        {
            var elements = new TLink[source.Length];
            for (int i = 0; i < source.Length; i++)
            {
                elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
            }
            _index.Add(elements);
            var sequence = _listToSequenceLinkConverter.Convert(elements);
            return Links.GetOrCreate(sequence, _unicodeSequenceMarker);
        }
    }
}
```

1.97 ./Platform.Data.Doublets/Unicode/UnicodeMap.cs

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Runtime.CompilerServices;
using System.Text;
using Platform.Data.Sequences;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Unicode
{
    public class UnicodeMap
    {
        public static readonly ulong FirstCharLink = 1;
        public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
        public static readonly ulong MapSize = 1 + char.MaxValue;

        private readonly ILinks<ulong> _links;
        private bool _initialized;

        public UnicodeMap(ILinks<ulong> links) => _links = links;

        public static UnicodeMap InitNew(ILinks<ulong> links)
        {
            var map = new UnicodeMap(links);
            map.Init();
            return map;
        }

        public void Init()
        {
            if (_initialized)
            {
                return;
            }
            _initialized = true;
            var firstLink = _links.CreatePoint();
            if (firstLink != FirstCharLink)
            {
                _links.Delete(firstLink);
            }
            else
            {
                for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
                {
                    // From NIL to It (NIL -> Character) transformation meaning, (or infinite
                    ↪  amount of NIL characters before actual Character)
                    var createdLink = _links.CreatePoint();
                    _links.Update(createdLink, firstLink, createdLink);
                    if (createdLink != i)
                    {
                        throw new InvalidOperationException("Unable to initialize UTF 16
                        ↪  table.");
                    }
                }
            }
        }

        // 0 - null link
        // 1 - nil character (0 character)
        // ...
        // 65536 (0(1) + 65535 = 65536 possible values)

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong FromCharToLink(char character) => (ulong)character + 1;

```

```
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          public static char FromLinkToChar(ulong link) => (char)(link - 1);
67
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          public static bool IsCharLink(ulong link) => link <= MapSize;
70
71          public static string FromLinksToString(IList<ulong> linksList)
72          {
73              var sb = new StringBuilder();
74              for (int i = 0; i < linksList.Count; i++)
75              {
76                  sb.Append(FromLinkToChar(linksList[i]));
77              }
78              return sb.ToString();
79          }
80
81          public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
82          {
83              var sb = new StringBuilder();
84              if (links.Exists(link))
85              {
86                  StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
87                      x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
                        ↪  element =>
88                      {
89                          sb.Append(FromLinkToChar(element));
90                          return true;
91                      });
92              }
93              return sb.ToString();
94          }
95
96          public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
            ↪  chars.Length);
97
98          public static ulong[] FromCharsToLinkArray(char[] chars, int count)
99          {
100             // char array to ulong array
101             var linksSequence = new ulong[count];
102             for (var i = 0; i < count; i++)
103             {
104                 linksSequence[i] = FromCharToLink(chars[i]);
105             }
106             return linksSequence;
107         }
108
109         public static ulong[] FromStringToLinkArray(string sequence)
110         {
111             // char array to ulong array
112             var linksSequence = new ulong[sequence.Length];
113             for (var i = 0; i < sequence.Length; i++)
114             {
115                 linksSequence[i] = FromCharToLink(sequence[i]);
116             }
117             return linksSequence;
118         }
119
120         public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
121         {
122             var result = new List<ulong[]>();
123             var offset = 0;
124             while (offset < sequence.Length)
125             {
126                 var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
127                 var relativeLength = 1;
128                 var absoluteLength = offset + relativeLength;
129                 while (absoluteLength < sequence.Length &&
130                        currentCategory ==
                           ↪  CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
131                 {
132                     relativeLength++;
133                     absoluteLength++;
134                 }
135                 // char array to ulong array
136                 var innerSequence = new ulong[relativeLength];
137                 var maxLength = offset + relativeLength;
138                 for (var i = offset; i < maxLength; i++)
139                 {
140                     innerSequence[i - offset] = FromCharToLink(sequence[i]);
```

```
                }
                result.Add(innerSequence);
                offset += relativeLength;
            }
            return result;
        }

        public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
        {
            var result = new List<ulong[]>();
            var offset = 0;
            while (offset < array.Length)
            {
                var relativeLength = 1;
                if (array[offset] <= LastCharLink)
                {
                    var currentCategory =
                    ↪ CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
                    var absoluteLength = offset + relativeLength;
                    while (absoluteLength < array.Length &&
                           array[absoluteLength] <= LastCharLink &&
                           currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
                           ↪ array[absoluteLength])))
                    {
                        relativeLength++;
                        absoluteLength++;
                    }
                }
                else
                {
                    var absoluteLength = offset + relativeLength;
                    while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
                    {
                        relativeLength++;
                        absoluteLength++;
                    }
                }
                // copy array
                var innerSequence = new ulong[relativeLength];
                var maxLength = offset + relativeLength;
                for (var i = offset; i < maxLength; i++)
                {
                    innerSequence[i - offset] = array[i];
                }
                result.Add(innerSequence);
                offset += relativeLength;
            }
            return result;
        }
    }
}
```

1.98 ./Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs

```
using Platform.Interfaces;
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Unicode
{
    public class UnicodeSequenceCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
    ↪ ICriterionMatcher<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;
        private readonly TLink _unicodeSequenceMarker;
        public UnicodeSequenceCriterionMatcher(ILinks<TLink> links, TLink unicodeSequenceMarker)
        ↪ : base(links) => _unicodeSequenceMarker = unicodeSequenceMarker;
        public bool IsMatched(TLink link) => _equalityComparer.Equals(Links.GetTarget(link),
        ↪ _unicodeSequenceMarker);
    }
}
```

1.99 ./Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs

```
using System;
using System.Linq;
using Platform.Interfaces;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Walkers;
```

```

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Unicode
{
    public class UnicodeSequenceToStringConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<TLink, string>
    {
        private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
        private readonly ISequenceWalker<TLink> _sequenceWalker;
        private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;

        public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
        ↪ unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
        ↪ IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
        {
            _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
            _sequenceWalker = sequenceWalker;
            _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
        }

        public string Convert(TLink source)
        {
            if(!_unicodeSequenceCriterionMatcher.IsMatched(source))
            {
                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                ↪ not a unicode sequence.");
            }
            var sequence = Links.GetSource(source);
            var charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.
            ↪ Convert).ToArray();
            return new string(charArray);
        }
    }
}
```

1.100 ./Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs

```
using Platform.Interfaces;
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Unicode
{
    public class UnicodeSymbolCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
    ↪ ICriterionMatcher<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ EqualityComparer<TLink>.Default;
        private readonly TLink _unicodeSymbolMarker;
        public UnicodeSymbolCriterionMatcher(ILinks<TLink> links, TLink unicodeSymbolMarker) :
        ↪ base(links) => _unicodeSymbolMarker = unicodeSymbolMarker;
        public bool IsMatched(TLink link) => _equalityComparer.Equals(Links.GetTarget(link),
        ↪ _unicodeSymbolMarker);
    }
}
```

1.101 ./Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs

```
using System;
using Platform.Interfaces;
using Platform.Converters;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Doublets.Unicode
{
    public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<TLink, char>
    {
        private static readonly UncheckedConverter<TLink, ushort> _addressToUInt16Converter =
        ↪ UncheckedConverter<TLink, ushort>.Default;

        private readonly IConverter<TLink> _numberToAddressConverter;
        private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;

        public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
        ↪ numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
        ↪ base(links)
        {
```

```
            _numberToAddressConverter = numberToAddressConverter;
            _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
        }

        public char Convert(TLink source)
        {
            if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
            {
                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                ↪  not a unicode symbol.");
            }
            return (char)_addressToUInt16Converter.Convert(_numberToAddressConverter.Convert(Lin
            ↪  ks.GetSource(source)));
        }
    }
}
```

1.102 ./Platform.Data.Doublets.Tests/ComparisonTests.cs

```
using System;
using System.Collections.Generic;
using Xunit;
using Platform.Diagnostics;

namespace Platform.Data.Doublets.Tests
{
    public static class ComparisonTests
    {
        private class UInt64Comparer : IComparer<ulong>
        {
            public int Compare(ulong x, ulong y) => x.CompareTo(y);
        }

        private static int Compare(ulong x, ulong y) => x.CompareTo(y);

        [Fact]
        public static void GreaterOrEqualPerfomanceTest()
        {
            const int N = 1000000;

            ulong x = 10;
            ulong y = 500;

            bool result = false;

            var ts1 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = Compare(x, y) >= 0;
                }
            });

            var comparer1 = Comparer<ulong>.Default;

            var ts2 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = comparer1.Compare(x, y) >= 0;
                }
            });

            Func<ulong, ulong, int> compareReference = comparer1.Compare;

            var ts3 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = compareReference(x, y) >= 0;
                }
            });

            var comparer2 = new UInt64Comparer();

            var ts4 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = comparer2.Compare(x, y) >= 0;
                }
```

```
            });

            Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {result}");
        }
    }
}
```

1.103 ./Platform.Data.Doublets.Tests/EqualityTests.cs

```
using System;
using System.Collections.Generic;
using Xunit;
using Platform.Diagnostics;

namespace Platform.Data.Doublets.Tests
{
    public static class EqualityTests
    {
        protected class UInt64EqualityComparer : IEqualityComparer<ulong>
        {
            public bool Equals(ulong x, ulong y) => x == y;

            public int GetHashCode(ulong obj) => obj.GetHashCode();
        }

        private static bool Equals1<T>(T x, T y) => Equals(x, y);

        private static bool Equals2<T>(T x, T y) => x.Equals(y);

        private static bool Equals3(ulong x, ulong y) => x == y;

        [Fact]
        public static void EqualsPerfomanceTest()
        {
            const int N = 1000000;

            ulong x = 10;
            ulong y = 500;

            bool result = false;

            var ts1 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = Equals1(x, y);
                }
            });

            var ts2 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = Equals2(x, y);
                }
            });

            var ts3 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = Equals3(x, y);
                }
            });

            var equalityComparer1 = EqualityComparer<ulong>.Default;

            var ts4 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = equalityComparer1.Equals(x, y);
                }
            });

            var equalityComparer2 = new UInt64EqualityComparer();

            var ts5 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
```

```
                    result = equalityComparer2.Equals(x, y);
                }
            });

            Func<ulong, ulong, bool> equalityComparer3 = equalityComparer2.Equals;

            var ts6 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = equalityComparer3(x, y);
                }
            });

            var comparer = Comparer<ulong>.Default;

            var ts7 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = comparer.Compare(x, y) == 0;
                }
            });

            Assert.True(ts2 < ts1);
            Assert.True(ts3 < ts2);
            Assert.True(ts5 < ts4);
            Assert.True(ts5 < ts6);

            Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {result}");
        }
    }
}
```

1.104 ./Platform.Data.Doublets.Tests/GenericLinksTests.cs

```
using System;
using Xunit;
using Platform.Reflection;
using Platform.Memory;
using Platform.Scopes;
using Platform.Data.Doublets.ResizableDirectMemory.Generic;

namespace Platform.Data.Doublets.Tests
{
    public unsafe static class GenericLinksTests
    {
        [Fact]
        public static void CRUDTest()
        {
            Using<byte>(links => links.TestCRUDOperations());
            Using<ushort>(links => links.TestCRUDOperations());
            Using<uint>(links => links.TestCRUDOperations());
            Using<ulong>(links => links.TestCRUDOperations());
        }

        [Fact]
        public static void RawNumbersCRUDTest()
        {
            Using<byte>(links => links.TestRawNumbersCRUDOperations());
            Using<ushort>(links => links.TestRawNumbersCRUDOperations());
            Using<uint>(links => links.TestRawNumbersCRUDOperations());
            Using<ulong>(links => links.TestRawNumbersCRUDOperations());
        }

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
            ↪  MultipleRandomCreationsAndDeletions(16)); // Cannot use more because current
            ↪  implementation of tree cuts out 5 bits from the address space.
            Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Te
            ↪  stMultipleRandomCreationsAndDeletions(100));
            Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Test
            ↪  MultipleRandomCreationsAndDeletions(100));
            Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
            ↪  tMultipleRandomCreationsAndDeletions(100));
        }

        private static void Using<TLink>(Action<ILinks<TLink>> action)
```

```
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  ResizableDirectMemoryLinks<TLink>>>())
            {
                action(scope.Use<ILinks<TLink>>());
            }
        }
    }
}
```

1.105 ./Platform.Data.Doublets.Tests/LinksConstantsTests.cs

```
using Xunit;

namespace Platform.Data.Doublets.Tests
{
    public static class LinksConstantsTests
    {
        [Fact]
        public static void ExternalReferencesTest()
        {
            LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
            ↪  (long.MaxValue + 1UL, ulong.MaxValue));

            //var minimum = new Hybrid<ulong>(0, isExternal: true);
            var minimum = new Hybrid<ulong>(1, isExternal: true);
            var maximum = new Hybrid<ulong>(long.MaxValue, isExternal: true);

            Assert.True(constants.IsExternalReference(minimum));
            Assert.True(constants.IsExternalReference(maximum));
        }
    }
}
```

1.106 ./Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs

```
using System;
using System.Linq;
using Xunit;
using Platform.Collections.Stacks;
using Platform.Collections.Arrays;
using Platform.Memory;
using Platform.Data.Numbers.Raw;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.PropertyOperators;
using Platform.Data.Doublets.Incrementers;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Unicode;
using Platform.Data.Doublets.Numbers.Unary;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.ResizableDirectMemory.Specific;

namespace Platform.Data.Doublets.Tests
{
    public static class OptimalVariantSequenceTests
    {
        private static readonly string _sequenceExample = "зеленела зелёная зелень";
        private static readonly string _loremIpsumExample = @"Lorem ipsum dolor sit amet,
        ↪  consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore
        ↪  magna aliqua.
Facilisi nullam vehicula ipsum a arcu cursus vitae congue mauris.
Et malesuada fames ac turpis egestas sed.
Eget velit aliquet sagittis id consectetur purus.
Dignissim cras tincidunt lobortis feugiat vivamus.
Vitae aliquet nec ullamcorper sit.
Lectus quam id leo in vitae.
Tortor dignissim convallis aenean et tortor at risus viverra adipiscing.
Sed risus ultricies tristique nulla aliquet enim tortor at auctor.
Integer eget aliquet nibh praesent tristique.
Vitae congue eu consequat ac felis donec et odio.
Tristique et egestas quis ipsum suspendisse.
Suspendisse potenti nullam ac tortor vitae purus faucibus ornare.
Nulla facilisi etiam dignissim diam quis enim lobortis scelerisque.
Imperdiet proin fermentum leo vel orci.
In ante metus dictum at tempor commodo.
Nisi lacus sed viverra tellus in.
Quam vulputate dignissim suspendisse in.
Elit scelerisque mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus.
Gravida cum sociis natoque penatibus et magnis dis parturient.
```

```
Risus quis varius quam quisque id diam.
Congue nisi vitae suscipit tellus mauris a diam maecenas.
Eget nunc scelerisque viverra mauris in aliquam sem fringilla.
Pharetra vel turpis nunc eget lorem dolor sed viverra.
Mattis pellentesque id nibh tortor id aliquet.
Purus non enim praesent elementum facilisis leo vel.
Etiam sit amet nisl purus in mollis nunc sed.
Tortor at auctor urna nunc id cursus metus aliquam.
Volutpat odio facilisis mauris sit amet.
Turpis egestas pretium aenean pharetra magna ac placerat.
Fermentum dui faucibus in ornare quam viverra orci sagittis eu.
Porttitor leo a diam sollicitudin tempor id eu.
Volutpat sed cras ornare arcu dui.
Ut aliquam purus sit amet luctus venenatis lectus magna.
Aliquet risus feugiat in ante metus dictum at.
Mattis nunc sed blandit libero.
Elit pellentesque habitant morbi tristique senectus et netus.
Nibh sit amet commodo nulla facilisi nullam vehicula ipsum a.
Enim sit amet venenatis urna cursus eget nunc scelerisque viverra.
Amet venenatis urna cursus eget nunc scelerisque viverra mauris in.
Diam donec adipiscing tristique risus nec feugiat.
Pulvinar mattis nunc sed blandit libero volutpat.
Cras fermentum odio eu feugiat pretium nibh ipsum.
In nulla posuere sollicitudin aliquam ultrices sagittis orci a.
Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.
A iaculis at erat pellentesque.
Morbi blandit cursus risus at ultrices mi tempus imperdiet nulla.
Eget lorem dolor sed viverra ipsum nunc.
Leo a diam sollicitudin tempor id eu.
Interdum consectetur libero id faucibus nisl tincidunt eget nullam non.";

        [Fact]
        public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: false))
            {
                var links = scope.Links;
                var constants = links.Constants;

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);

                var meaningRoot = links.CreatePoint();
                var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
                var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
                var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                ↪  constants.Itself);

                var unaryNumberToAddressConverter = new
                ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
                var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
                var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
                var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                ↪  frequencyPropertyMarker, frequencyMarker);
                var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                ↪  frequencyPropertyOperator, frequencyIncrementer);
                var linkToItsFrequencyNumberConverter = new
                ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                ↪  unaryNumberToAddressConverter);
                var sequenceToItsLocalElementLevelsConverter = new
                ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                ↪  linkToItsFrequencyNumberConverter);
                var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                ↪  sequenceToItsLocalElementLevelsConverter);

                var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
                ↪  Walker = new LeveledSequenceWalker<ulong>(links) });

                ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
                ↪  index, optimalVariantConverter);
            }
        }

        [Fact]
        public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: false))
            {
```

```
114                 var links = scope.Links;
115
116                 links.UseUnicode();
117
118                 var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
119
120                 var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links);
121
122                 var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
                    ↪  totalSequenceSymbolFrequencyCounter);
123
124                 var index = new
                    ↪  CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
125                 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<ulong>(linkFrequenciesCache);
126
127                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
128                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
129
130                 var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
                    ↪  Walker = new LeveledSequenceWalker<ulong>(links) });
131
132                 ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
                    ↪  index, optimalVariantConverter);
133             }
134         }
135
136         private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
            ↪  SequenceToItsLocalElementLevelsConverter<ulong>
            ↪  sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
            ↪  OptimalVariantConverter<ulong> optimalVariantConverter)
137         {
138             index.Add(sequence);
139
140             var optimalVariant = optimalVariantConverter.Convert(sequence);
141
142             var readSequence1 = sequences.ToList(optimalVariant);
143
144             Assert.True(sequence.SequenceEqual(readSequence1));
145         }
146
147         [Fact]
148         public static void SavedSequencesOptimizationTest()
149         {
150             LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
                    ↪  (long.MaxValue + 1UL, ulong.MaxValue));
151
152             using (var memory = new HeapResizableDirectMemory())
153             using (var disposableLinks = new UInt64ResizableDirectMemoryLinks(memory,
                    ↪  UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep, constants,
                    ↪  useAvlBasedIndex: false))
154             {
155                 var links = new UInt64Links(disposableLinks);
156
157                 var root = links.CreatePoint();
158
159                 //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
160                 var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
161
162                 var unicodeSymbolMarker = links.GetOrCreate(root,
                    ↪  addressToNumberConverter.Convert(1));
163                 var unicodeSequenceMarker = links.GetOrCreate(root,
                    ↪  addressToNumberConverter.Convert(2));
164
165                 var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links);
166                 var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
                    ↪  totalSequenceSymbolFrequencyCounter);
167                 var index = new
                    ↪  CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
168                 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<ulong>(linkFrequenciesCache);
```

```
                var sequenceToItsLocalElementLevelsConverter = new
                ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                ↪  linkToItsFrequencyNumberConverter);
                var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                ↪  sequenceToItsLocalElementLevelsConverter);

                var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
                ↪  (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));

                var unicodeSequencesOptions = new SequencesOptions<ulong>()
                {
                    UseSequenceMarker = true,
                    SequenceMarkerLink = unicodeSequenceMarker,
                    UseIndex = true,
                    Index = index,
                    LinksToSequenceConverter = optimalVariantConverter,
                    Walker = walker,
                    UseGarbageCollection = true
                };

                var unicodeSequences = new Sequences.Sequences(new
                ↪  SynchronizedLinks<ulong>(links), unicodeSequencesOptions);

                // Create some sequences
                var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
                ↪  StringSplitOptions.RemoveEmptyEntries);
                var arrays = strings.Select(x => x.Select(y =>
                ↪  addressToNumberConverter.Convert(y)).ToArray()).ToArray();
                for (int i = 0; i < arrays.Length; i++)
                {
                    unicodeSequences.Create(arrays[i].ShiftRight());
                }

                var linksCountAfterCreation = links.Count();

                // get list of sequences links
                // for each sequence link
                //   create new sequence version
                //   if new sequence is not the same as sequence link
                //     delete sequence link
                //     collect garbadge
                unicodeSequences.CompactAll();

                var linksCountAfterCompactification = links.Count();

                Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
            }
        }
    }
}
```

1.107 ./Platform.Data.Doublets.Tests/ReadSequenceTests.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using Xunit;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Sequences;

namespace Platform.Data.Doublets.Tests
{
    public static class ReadSequenceTests
    {
        [Fact]
        public static void ReadSequenceTest()
        {
            const long sequenceLength = 2000;

            using (var scope = new TempLinksTestScope(useSequences: false))
            {
                var links = scope.Links;
                var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong> {
                ↪  Walker = new LeveledSequenceWalker<ulong>(links) });

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
```

```
                    sequence[i] = links.Create();
                }

                var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);

                var sw1 = Stopwatch.StartNew();
                var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();

                var sw2 = Stopwatch.StartNew();
                var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();

                var sw3 = Stopwatch.StartNew();
                var readSequence2 = new List<ulong>();
                SequenceWalker.WalkRight(balancedVariant,
                                         links.GetSource,
                                         links.GetTarget,
                                         links.IsPartialPoint,
                                         readSequence2.Add);
                sw3.Stop();

                Assert.True(sequence.SequenceEqual(readSequence1));

                Assert.True(sequence.SequenceEqual(readSequence2));

                // Assert.True(sw2.Elapsed < sw3.Elapsed);

                Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
                ↪  {sw2.Elapsed}");

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }
            }
        }
    }
}
```

1.108 ./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs

```
using System.IO;
using Xunit;
using Platform.Singletons;
using Platform.Memory;
using Platform.Data.Doublets.ResizableDirectMemory.Specific;

namespace Platform.Data.Doublets.Tests
{
    public static class ResizableDirectMemoryLinksTests
    {
        private static readonly LinksConstants<ulong> _constants =
        ↪  Default<LinksConstants<ulong>>.Instance;

        [Fact]
        public static void BasicFileMappedMemoryTest()
        {
            var tempFilename = Path.GetTempFileName();
            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(tempFilename))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
            File.Delete(tempFilename);
        }

        [Fact]
        public static void BasicHeapMemoryTest()
        {
            using (var memory = new
            ↪  HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
            ↪  UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestBasicMemoryOperations();
            }
        }

        private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
        {
            var link = memoryAdapter.Create();
            memoryAdapter.Delete(link);
```

```
        }

        [Fact]
        public static void NonexistentReferencesHeapMemoryTest()
        {
            using (var memory = new
            ↪  HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
            ↪  UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
            {
                memoryAdapter.TestNonexistentReferences();
            }
        }

        private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
        {
            var link = memoryAdapter.Create();
            memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
            var resultLink = _constants.Null;
            memoryAdapter.Each(foundLink =>
            {
                resultLink = foundLink[_constants.IndexPart];
                return _constants.Break;
            }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
            Assert.True(resultLink == link);
            Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
            memoryAdapter.Delete(link);
        }
    }
}
```

1.109 ./Platform.Data.Doublets.Tests/ScopeTests.cs

```
using Xunit;
using Platform.Scopes;
using Platform.Memory;
using Platform.Data.Doublets.Decorators;
using Platform.Reflection;
using Platform.Data.Doublets.ResizableDirectMemory.Generic;
using Platform.Data.Doublets.ResizableDirectMemory.Specific;

namespace Platform.Data.Doublets.Tests
{
    public static class ScopeTests
    {
        [Fact]
        public static void SingleDependencyTest()
        {
            using (var scope = new Scope())
            {
                scope.IncludeAssemblyOf<IMemory>();
                var instance = scope.Use<IDirectMemory>();
                Assert.IsType<HeapResizableDirectMemory>(instance);
            }
        }

        [Fact]
        public static void CascadeDependencyTest()
        {
            using (var scope = new Scope())
            {
                scope.Include<TemporaryFileMappedResizableDirectMemory>();
                scope.Include<UInt64ResizableDirectMemoryLinks>();
                var instance = scope.Use<ILinks<ulong>>();
                Assert.IsType<UInt64ResizableDirectMemoryLinks>(instance);
            }
        }

        [Fact]
        public static void FullAutoResolutionTest()
        {
            using (var scope = new Scope(autoInclude: true, autoExplore: true))
            {
                var instance = scope.Use<UInt64Links>();
                Assert.IsType<UInt64Links>(instance);
            }
        }

        [Fact]
        public static void TypeParametersTest()
```

```
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  ResizableDirectMemoryLinks<ulong>>>())
            {
                var links = scope.Use<ILinks<ulong>>();
                Assert.IsType<ResizableDirectMemoryLinks<ulong>>(links);
            }
        }
    }
}
```

1.110 ./Platform.Data.Doublets.Tests/SequencesTests.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using Xunit;
using Platform.Collections;
using Platform.Collections.Arrays;
using Platform.Random;
using Platform.IO;
using Platform.Singletons;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Unicode;

namespace Platform.Data.Doublets.Tests
{
    public static class SequencesTests
    {
        private static readonly LinksConstants<ulong> _constants =
        ↪  Default<LinksConstants<ulong>>.Instance;

        static SequencesTests()
        {
            // Trigger static constructor to not mess with perfomance measurements
            _ = BitString.GetBitMaskFromIndex(1);
        }

        [Fact]
        public static void CreateAllVariantsTest()
        {
            const long sequenceLength = 8;

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
                    sequence[i] = links.Create();
                }

                var sw1 = Stopwatch.StartNew();
                var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();

                var sw2 = Stopwatch.StartNew();
                var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();

                Assert.True(results1.Count > results2.Length);
                Assert.True(sw1.Elapsed > sw2.Elapsed);

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }

                Assert.True(links.Count() == 0);
            }
        }

        //[Fact]
        //public void CUDTest()
        //{
        //    var tempFilename = Path.GetTempFileName();

```

```
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
```

```
            var intersection3 = createResults.Intersect(searchResults3).ToList();
            Assert.True(intersection3.Count == searchResults3.Count);
            Assert.True(intersection3.Count == createResults.Length);

            for (var i = 0; i < sequenceLength; i++)
            {
                links.Delete(sequence[i]);
            }
        }
    }

    [Fact]
    public static void BalancedVariantSearchTest()
    {
        const long sequenceLength = 200;

        using (var scope = new TempLinksTestScope(useSequences: true))
        {
            var links = scope.Links;
            var sequences = scope.Sequences;

            var sequence = new ulong[sequenceLength];
            for (var i = 0; i < sequenceLength; i++)
            {
                sequence[i] = links.Create();
            }

            var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);

            var sw1 = Stopwatch.StartNew();
            var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();

            var sw2 = Stopwatch.StartNew();
            var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();

            var sw3 = Stopwatch.StartNew();
            var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();

            // На количестве в 200 элементов это будет занимать вечность
            //var sw4 = Stopwatch.StartNew();
            //var searchResults4 = sequences.Each(sequence); sw4.Stop();

            Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);

            Assert.True(searchResults3.Count == 1 && balancedVariant ==
            ↪  searchResults3.First());

            //Assert.True(sw1.Elapsed < sw2.Elapsed);

            for (var i = 0; i < sequenceLength; i++)
            {
                links.Delete(sequence[i]);
            }
        }
    }

    [Fact]
    public static void AllPartialVariantsSearchTest()
    {
        const long sequenceLength = 8;

        using (var scope = new TempLinksTestScope(useSequences: true))
        {
            var links = scope.Links;
            var sequences = scope.Sequences;

            var sequence = new ulong[sequenceLength];
            for (var i = 0; i < sequenceLength; i++)
            {
                sequence[i] = links.Create();
            }

            var createResults = sequences.CreateAllVariants2(sequence);

            //var createResultsStrings = createResults.Select(x => x + ": " +
            ↪  sequences.FormatSequence(x)).ToList();
            //Global.Trash = createResultsStrings;

            var partialSequence = new ulong[sequenceLength - 2];

```

```
225                 Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
226
227                 var sw1 = Stopwatch.StartNew();
228                 var searchResults1 =
                    ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
229
230                 var sw2 = Stopwatch.StartNew();
231                 var searchResults2 =
                    ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
232
233                 //var sw3 = Stopwatch.StartNew();
234                 //var searchResults3 =
                    ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
235
236                 var sw4 = Stopwatch.StartNew();
237                 var searchResults4 =
                    ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
238
239                 //Global.Trash = searchResults3;
240
241                 //var searchResults1Strings = searchResults1.Select(x => x + ": " +
                    ↪ sequences.FormatSequence(x)).ToList();
242                 //Global.Trash = searchResults1Strings;
243
244                 var intersection1 = createResults.Intersect(searchResults1).ToList();
245                 Assert.True(intersection1.Count == createResults.Length);
246
247                 var intersection2 = createResults.Intersect(searchResults2).ToList();
248                 Assert.True(intersection2.Count == createResults.Length);
249
250                 var intersection4 = createResults.Intersect(searchResults4).ToList();
251                 Assert.True(intersection4.Count == createResults.Length);
252
253                 for (var i = 0; i < sequenceLength; i++)
254                 {
255                     links.Delete(sequence[i]);
256                 }
257             }
258         }
259
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 =
                    ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
286
287                 var sw2 = Stopwatch.StartNew();
288                 var searchResults2 =
                    ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
289
290                 Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
291
292                 Assert.True(searchResults2.Count == 1 && balancedVariant ==
                    ↪ searchResults2.First());
293
294                 for (var i = 0; i < sequenceLength; i++)
295                 {
```

```
                links.Delete(sequence[i]);
            }
        }
    }

    [Fact(Skip = "Correct implementation is pending")]
    public static void PatternMatchTest()
    {
        var zeroOrMany = Sequences.Sequences.ZeroOrMany;

        using (var scope = new TempLinksTestScope(useSequences: true))
        {
            var links = scope.Links;
            var sequences = scope.Sequences;

            var e1 = links.Create();
            var e2 = links.Create();

            var sequence = new[]
            {
                e1, e2, e1, e2 // mama / papa
            };

            var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);

            var balancedVariant = balancedVariantConverter.Convert(sequence);

            // 1: [1]
            // 2: [2]
            // 3: [1,2]
            // 4: [1,2,1,2]

            var doublet = links.GetSource(balancedVariant);

            var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);

            Assert.True(matchedSequences1.Count == 0);

            var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);

            Assert.True(matchedSequences2.Count == 0);

            var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);

            Assert.True(matchedSequences3.Count == 0);

            var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);

            Assert.Contains(doublet, matchedSequences4);
            Assert.Contains(balancedVariant, matchedSequences4);

            for (var i = 0; i < sequence.Length; i++)
            {
                links.Delete(sequence[i]);
            }
        }
    }

    [Fact]
    public static void IndexTest()
    {
        using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
        ↪  true }, useSequences: true))
        {
            var links = scope.Links;
            var sequences = scope.Sequences;
            var index = sequences.Options.Index;

            var e1 = links.Create();
            var e2 = links.Create();

            var sequence = new[]
            {
                e1, e2, e1, e2 // mama / papa
            };

            Assert.False(index.MightContain(sequence));

            index.Add(sequence);

```

```
                Assert.True(index.MightContain(sequence));
            }
        }

        /// <summary>Imported from https://raw.githubusercontent.com/wiki/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>
        private static readonly string _exampleText =
            @"([english version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))

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

[![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?

[![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)

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?

[![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)

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?

[![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)

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?

[![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)

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?

[![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)
```

```
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?

[![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)

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?

[![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)

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?

[![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)

На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом рекурсии или фрактала?

[![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)

Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы? Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?

[![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)

...

[![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif)";

        private static readonly string _exampleLoremIpsumText =
            @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.";

        [Fact]
        public static void CompressionTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var e1 = links.Create();
                var e2 = links.Create();

                var sequence = new[]
                {
                    e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
```

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

```
517                 //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                    ↪  constants.Itself);
518
519                 //var unaryNumberToAddressConverter = new
                    ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
520                 //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
                    ↪  unaryOne);
521                 //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
522                 //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
                    ↪  frequencyPropertyMarker, frequencyMarker);
523                 //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
524                 //var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
525
526                 var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
527
528                 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<ulong>(linkFrequenciesCache3);
529
530                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
                    ↪  linkToItsFrequencyNumberConverter);
531                 var optimalVariantConverter = new
                    ↪  OptimalVariantConverter<ulong>(scope3.Links.Unsync,
                    ↪  sequenceToItsLocalElementLevelsConverter);
532
533                 var compressed1 = new ulong[arrays.Length];
534                 var compressed2 = new ulong[arrays.Length];
535                 var compressed3 = new ulong[arrays.Length];
536
537                 var START = 0;
538                 var END = arrays.Length;
539
540                 //for (int i = START; i < END; i++)
541                 //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
542
543                 var initialCount1 = scope2.Links.Unsync.Count();
544
545                 var sw1 = Stopwatch.StartNew();
546
547                 for (int i = START; i < END; i++)
548                 {
549                     linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
550                     compressed1[i] = compressor1.Convert(arrays[i]);
551                 }
552
553                 var elapsed1 = sw1.Elapsed;
554
555                 var balancedVariantConverter2 = new
                    ↪  BalancedVariantConverter<ulong>(scope2.Links.Unsync);
556
557                 var initialCount2 = scope2.Links.Unsync.Count();
558
559                 var sw2 = Stopwatch.StartNew();
560
561                 for (int i = START; i < END; i++)
562                 {
563                     compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
564                 }
565
566                 var elapsed2 = sw2.Elapsed;
567
568                 for (int i = START; i < END; i++)
569                 {
570                     linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
571                 }
572
573                 var initialCount3 = scope3.Links.Unsync.Count();
574
575                 var sw3 = Stopwatch.StartNew();
576
577                 for (int i = START; i < END; i++)
578                 {
579                     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
580                     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
```

```
                }

                var elapsed3 = sw3.Elapsed;

                Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
                ↪  Optimal variant: {elapsed3}");

                // Assert.True(elapsed1 > elapsed2);

                // Checks
                for (int i = START; i < END; i++)
                {
                    var sequence1 = compressed1[i];
                    var sequence2 = compressed2[i];
                    var sequence3 = compressed3[i];

                    var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                    ↪  scope1.Links.Unsync);

                    var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                    ↪  scope2.Links.Unsync);

                    var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
                    ↪  scope3.Links.Unsync);

                    var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
                    ↪  link.IsPartialPoint());
                    var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
                    ↪  link.IsPartialPoint());
                    var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
                    ↪  link.IsPartialPoint());

                    //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                    ↪  arrays[i].Length > 3)
                    //    Assert.False(structure1 == structure2);
                    //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
                    ↪  arrays[i].Length > 3)
                    //    Assert.False(structure3 == structure2);

                    Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
                    Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
                }

                Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
                ↪  totalCharacters);
                Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
                ↪  totalCharacters);
                Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
                ↪  totalCharacters);

                Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
                ↪  totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
                ↪  totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
                ↪  totalCharacters}");

                Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
                ↪  scope2.Links.Unsync.Count() - initialCount2);
                Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
                ↪  scope2.Links.Unsync.Count() - initialCount2);

                var duplicateProvider1 = new
                ↪  DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
                var duplicateProvider2 = new
                ↪  DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
                var duplicateProvider3 = new
                ↪  DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);

                var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
                var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
                var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);

                var duplicates1 = duplicateCounter1.Count();

                ConsoleHelpers.Debug("------");

                var duplicates2 = duplicateCounter2.Count();

                ConsoleHelpers.Debug("------");
```

```
639
640                 var duplicates3 = duplicateCounter3.Count();
641
642                 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
643
644                 linkFrequenciesCache1.ValidateFrequencies();
645                 linkFrequenciesCache3.ValidateFrequencies();
646             }
647         }
648
649         [Fact]
650         public static void CompressionStabilityTest()
651         {
652             // TODO: Fix bug (do a separate test)
653             //const ulong minNumbers = 0;
654             //const ulong maxNumbers = 1000;
655
656             const ulong minNumbers = 10000;
657             const ulong maxNumbers = 12500;
658
659             var strings = new List<string>();
660
661             for (ulong i = minNumbers; i < maxNumbers; i++)
662             {
663                 strings.Add(i.ToString());
664             }
665
666             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
667             var totalCharacters = arrays.Select(x => x.Length).Sum();
668
669             using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
                 ↪ SequencesOptions<ulong> { UseCompression = true,
                 ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
670             using (var scope2 = new TempLinksTestScope(useSequences: true))
671             {
672                 scope1.Links.UseUnicode();
673                 scope2.Links.UseUnicode();
674
675                 //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
676                 var compressor1 = scope1.Sequences;
677                 var compressor2 = scope2.Sequences;
678
679                 var compressed1 = new ulong[arrays.Length];
680                 var compressed2 = new ulong[arrays.Length];
681
682                 var sw1 = Stopwatch.StartNew();
683
684                 var START = 0;
685                 var END = arrays.Length;
686
687                 // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
688                 // Stability issue starts at 10001 or 11000
689                 //for (int i = START; i < END; i++)
690                 //{
691                 //    var first = compressor1.Compress(arrays[i]);
692                 //    var second = compressor1.Compress(arrays[i]);
693
694                 //    if (first == second)
695                 //        compressed1[i] = first;
696                 //    else
697                 //    {
698                 //        // TODO: Find a solution for this case
699                 //    }
700                 //}
701
702                 for (int i = START; i < END; i++)
703                 {
704                     var first = compressor1.Create(arrays[i].ShiftRight());
705                     var second = compressor1.Create(arrays[i].ShiftRight());
706
707                     if (first == second)
708                     {
709                         compressed1[i] = first;
710                     }
711                     else
712                     {
713                         // TODO: Find a solution for this case
714                     }
715                 }
```

```
716
717                 var elapsed1 = sw1.Elapsed;
718
719                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
720
721                 var sw2 = Stopwatch.StartNew();
722
723                 for (int i = START; i < END; i++)
724                 {
725                     var first = balancedVariantConverter.Convert(arrays[i]);
726                     var second = balancedVariantConverter.Convert(arrays[i]);
727
728                     if (first == second)
729                     {
730                         compressed2[i] = first;
731                     }
732                 }
733
734                 var elapsed2 = sw2.Elapsed;
735
736                 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                    {elapsed2}");
737
738                 Assert.True(elapsed1 > elapsed2);
739
740                 // Checks
741                 for (int i = START; i < END; i++)
742                 {
743                     var sequence1 = compressed1[i];
744                     var sequence2 = compressed2[i];
745
746                     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
747                     {
748                         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            scope1.Links);
749
750                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            scope2.Links);
751
752                         //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
                            link.IsPartialPoint());
753                         //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
                            link.IsPartialPoint());
754
755                         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                            arrays[i].Length > 3)
756                         //    Assert.False(structure1 == structure2);
757
758                         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
759                     }
760                 }
761
762                 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
763                 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
764
765                 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                    totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                    totalCharacters}");
766
767                 Assert.True(scope1.Links.Count() <= scope2.Links.Count());
768
769                 //compressor1.ValidateFrequencies();
770             }
771         }
772
773         [Fact]
774         public static void RundomNumbersCompressionQualityTest()
775         {
776             const ulong N = 500;
777
778             //const ulong minNumbers = 10000;
779             //const ulong maxNumbers = 20000;
780
781             //var strings = new List<string>();
782
783             //for (ulong i = 0; i < N; i++)
784             //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
                    maxNumbers).ToString());
785
```

```
            var strings = new List<string>();

            for (ulong i = 0; i < N; i++)
            {
                strings.Add(RandomHelpers.Default.NextUInt64().ToString());
            }

            strings = strings.Distinct().ToList();

            var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
            var totalCharacters = arrays.Select(x => x.Length).Sum();

            using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
            ↪  SequencesOptions<ulong> { UseCompression = true,
            ↪  EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
            using (var scope2 = new TempLinksTestScope(useSequences: true))
            {
                scope1.Links.UseUnicode();
                scope2.Links.UseUnicode();

                var compressor1 = scope1.Sequences;
                var compressor2 = scope2.Sequences;

                var compressed1 = new ulong[arrays.Length];
                var compressed2 = new ulong[arrays.Length];

                var sw1 = Stopwatch.StartNew();

                var START = 0;
                var END = arrays.Length;

                for (int i = START; i < END; i++)
                {
                    compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
                }

                var elapsed1 = sw1.Elapsed;

                var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);

                var sw2 = Stopwatch.StartNew();

                for (int i = START; i < END; i++)
                {
                    compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
                }

                var elapsed2 = sw2.Elapsed;

                Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                ↪  {elapsed2}");

                Assert.True(elapsed1 > elapsed2);

                // Checks
                for (int i = START; i < END; i++)
                {
                    var sequence1 = compressed1[i];
                    var sequence2 = compressed2[i];

                    if (sequence1 != _constants.Null && sequence2 != _constants.Null)
                    {
                        var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                        ↪  scope1.Links);

                        var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                        ↪  scope2.Links);

                        Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
                    }
                }

                Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
                Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);

                Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                ↪  totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                ↪  totalCharacters}");

```

```
                // Can be worse than balanced variant
                //Assert.True(scope1.Links.Count() <= scope2.Links.Count());

                //compressor1.ValidateFrequencies();
            }
        }

        [Fact]
        public static void AllTreeBreakDownAtSequencesCreationBugTest()
        {
            // Made out of AllPossibleConnectionsTest test.

            //const long sequenceLength = 5; //100% bug
            const long sequenceLength = 4; //100% bug
            //const long sequenceLength = 3; //100% _no_bug (ok)

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
                    sequence[i] = links.Create();
                }

                var createResults = sequences.CreateAllVariants2(sequence);

                Global.Trash = createResults;

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }
            }
        }

        [Fact]
        public static void AllPossibleConnectionsTest()
        {
            const long sequenceLength = 5;

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
                    sequence[i] = links.Create();
                }

                var createResults = sequences.CreateAllVariants2(sequence);
                var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());

                for (var i = 0; i < 1; i++)
                {
                    var sw1 = Stopwatch.StartNew();
                    var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();

                    var sw2 = Stopwatch.StartNew();
                    var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();

                    var sw3 = Stopwatch.StartNew();
                    var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();

                    var sw4 = Stopwatch.StartNew();
                    var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();

                    Global.Trash = searchResults3;
                    Global.Trash = searchResults4; //-V3008

                    var intersection1 = createResults.Intersect(searchResults1).ToList();
                    Assert.True(intersection1.Count == createResults.Length);

                    var intersection2 = reverseResults.Intersect(searchResults1).ToList();
                    Assert.True(intersection2.Count == reverseResults.Length);

```

```
                    var intersection0 = searchResults1.Intersect(searchResults2).ToList();
                    Assert.True(intersection0.Count == searchResults2.Count);

                    var intersection3 = searchResults2.Intersect(searchResults3).ToList();
                    Assert.True(intersection3.Count == searchResults3.Count);

                    var intersection4 = searchResults3.Intersect(searchResults4).ToList();
                    Assert.True(intersection4.Count == searchResults4.Count);
                }

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }
            }
        }

        [Fact(Skip = "Correct implementation is pending")]
        public static void CalculateAllUsagesTest()
        {
            const long sequenceLength = 3;

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
                    sequence[i] = links.Create();
                }

                var createResults = sequences.CreateAllVariants2(sequence);

                //var reverseResults =
                ↪  sequences.CreateAllVariants2(sequence.Reverse().ToArray());

                for (var i = 0; i < 1; i++)
                {
                    var linksTotalUsages1 = new ulong[links.Count() + 1];

                    sequences.CalculateAllUsages(linksTotalUsages1);

                    var linksTotalUsages2 = new ulong[links.Count() + 1];

                    sequences.CalculateAllUsages2(linksTotalUsages2);

                    var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
                    Assert.True(intersection1.Count == linksTotalUsages2.Length);
                }

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }
            }
        }
    }
}
```

1.111 ./Platform.Data.Doublets.Tests/TempLinksTestScope.cs

```
using System.IO;
using Platform.Disposables;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.ResizableDirectMemory.Specific;

namespace Platform.Data.Doublets.Tests
{
    public class TempLinksTestScope : DisposableBase
    {
        public ILinks<ulong> MemoryAdapter { get; }
        public SynchronizedLinks<ulong> Links { get; }
        public Sequences.Sequences Sequences { get; }
        public string TempFilename { get; }
        public string TempTransactionLogFilename { get; }
        private readonly bool _deleteFiles;

```

```
        public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
        ↪  useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
        ↪  useLog) { }

        public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
        ↪  true, bool useSequences = false, bool useLog = false)
        {
            _deleteFiles = deleteFiles;
            TempFilename = Path.GetTempFileName();
            TempTransactionLogFilename = Path.GetTempFileName();
            var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);
            MemoryAdapter = useLog ? (ILinks<ulong>)new
            ↪  UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
            ↪  coreMemoryAdapter;
            Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
            if (useSequences)
            {
                Sequences = new Sequences.Sequences(Links, sequencesOptions);
            }
        }

        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                Links.Unsync.DisposeIfPossible();
                if (_deleteFiles)
                {
                    DeleteFiles();
                }
            }
        }

        public void DeleteFiles()
        {
            File.Delete(TempFilename);
            File.Delete(TempTransactionLogFilename);
        }
    }
}
```

1.112 ./Platform.Data.Doublets.Tests/TestExtensions.cs

```
using System.Collections.Generic;
using Xunit;
using Platform.Ranges;
using Platform.Numbers;
using Platform.Random;
using Platform.Setters;
using Platform.Converters;

namespace Platform.Data.Doublets.Tests
{
    public static class TestExtensions
    {
        public static void TestCRUDOperations<T>(this ILinks<T> links)
        {
            var constants = links.Constants;

            var equalityComparer = EqualityComparer<T>.Default;

            var zero = default(T);
            var one = Arithmetic.Increment(zero);

            // Create Link
            Assert.True(equalityComparer.Equals(links.Count(), zero));

            var setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);

            Assert.True(equalityComparer.Equals(setter.Result, constants.Null));

            var linkAddress = links.Create();

            var link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(link.Count == 3);
            Assert.True(equalityComparer.Equals(link.Index, linkAddress));
            Assert.True(equalityComparer.Equals(link.Source, constants.Null));
            Assert.True(equalityComparer.Equals(link.Target, constants.Null));

```

```
            Assert.True(equalityComparer.Equals(links.Count(), one));

            // Get first link
            setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);

            Assert.True(equalityComparer.Equals(setter.Result, linkAddress));

            // Update link to reference itself
            links.Update(linkAddress, linkAddress, linkAddress);

            link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(equalityComparer.Equals(link.Source, linkAddress));
            Assert.True(equalityComparer.Equals(link.Target, linkAddress));

            // Update link to reference null (prepare for delete)
            var updated = links.Update(linkAddress, constants.Null, constants.Null);

            Assert.True(equalityComparer.Equals(updated, linkAddress));

            link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(equalityComparer.Equals(link.Source, constants.Null));
            Assert.True(equalityComparer.Equals(link.Target, constants.Null));

            // Delete link
            links.Delete(linkAddress);

            Assert.True(equalityComparer.Equals(links.Count(), zero));

            setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);

            Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
        }

        public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
        {
            // Constants
            var constants = links.Constants;
            var equalityComparer = EqualityComparer<T>.Default;

            var zero = default(T);
            var one = Arithmetic.Increment(zero);
            var two = Arithmetic.Increment(one);

            var h106E = new Hybrid<T>(106L, isExternal: true);
            var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
            var h108E = new Hybrid<T>(-108L);

            Assert.Equal(106L, h106E.AbsoluteValue);
            Assert.Equal(107L, h107E.AbsoluteValue);
            Assert.Equal(108L, h108E.AbsoluteValue);

            // Create Link (External -> External)
            var linkAddress1 = links.Create();

            links.Update(linkAddress1, h106E, h108E);

            var link1 = new Link<T>(links.GetLink(linkAddress1));

            Assert.True(equalityComparer.Equals(link1.Source, h106E));
            Assert.True(equalityComparer.Equals(link1.Target, h108E));

            // Create Link (Internal -> External)
            var linkAddress2 = links.Create();

            links.Update(linkAddress2, linkAddress1, h108E);

            var link2 = new Link<T>(links.GetLink(linkAddress2));

            Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
            Assert.True(equalityComparer.Equals(link2.Target, h108E));

            // Create Link (Internal -> Internal)
            var linkAddress3 = links.Create();

            links.Update(linkAddress3, linkAddress1, linkAddress2);

```

```
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))
```

```
                    {
                        links.Delete(link);
                        deleted++;
                    }
                }
                Assert.True(addressToUInt64Converter.Convert(links.Count()) == 0L);
            }
        }
    }
}
```

1.113 ./Platform.Data.Doublets.Tests/UInt64LinksTests.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Xunit;
using Platform.Disposables;
using Platform.Ranges;
using Platform.Random;
using Platform.Timestamps;
using Platform.Reflection;
using Platform.Singletons;
using Platform.Scopes;
using Platform.Counters;
using Platform.Diagnostics;
using Platform.IO;
using Platform.Memory;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.ResizableDirectMemory.Specific;

namespace Platform.Data.Doublets.Tests
{
    public static class UInt64LinksTests
    {
        private static readonly LinksConstants<ulong> _constants =
        ↪  Default<LinksConstants<ulong>>.Instance;

        private const long Iterations = 10 * 1024;

        #region Concept

        [Fact]
        public static void MultipleCreateAndDeleteTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  UInt64ResizableDirectMemoryLinks>>())
            {
                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

```

```
66                  Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
                    ↪  e.TempTransactionLogFilename);
67              }
68          }
69
70          [Fact]
71          public static void BasicTransactionLogTest()
72          {
73              using (var scope = new TempLinksTestScope(useLog: true))
74              {
75                  var links = scope.Links;
76                  var l1 = links.Create();
77                  var l2 = links.Create();
78
79                  Global.Trash = links.Update(l2, l2, l1, l2);
80
81                  links.Delete(l1);
82
83                  links.Unsync.DisposeIfPossible(); // Close links to access log
84
85                  Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
                    ↪  e.TempTransactionLogFilename);
86              }
87          }
88
89          [Fact]
90          public static void TransactionAutoRevertedTest()
91          {
92              // Auto Reverted (Because no commit at transaction)
93              using (var scope = new TempLinksTestScope(useLog: true))
94              {
95                  var links = scope.Links;
96                  var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
97                  using (var transaction = transactionsLayer.BeginTransaction())
98                  {
99                      var l1 = links.Create();
100                     var l2 = links.Create();
101
102                     links.Update(l2, l2, l1, l2);
103                 }
104
105                 Assert.Equal(0UL, links.Count());
106
107                 links.Unsync.DisposeIfPossible();
108
109                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
                    ↪  cope.TempTransactionLogFilename);
110                 Assert.Single(transitions);
111             }
112         }
113
114         [Fact]
115         public static void TransactionUserCodeErrorNoDataSavedTest()
116         {
117             // User Code Error (Autoreverted), no data saved
118             var itself = _constants.Itself;
119
120             TempLinksTestScope lastScope = null;
121             try
122             {
123                 using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                    ↪  useLog: true))
124                 {
125                     var links = scope.Links;
126                     var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
                        ↪  atorBase<ulong>)links.Unsync).Links;
127                     using (var transaction = transactionsLayer.BeginTransaction())
128                     {
129                         var l1 = links.CreateAndUpdate(itself, itself);
130                         var l2 = links.CreateAndUpdate(itself, itself);
131
132                         l2 = links.Update(l2, l2, l1, l2);
133
134                         links.CreateAndUpdate(l2, itself);
135                         links.CreateAndUpdate(l2, itself);
136
137                         //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
                            ↪  tion>(scope.TempTransactionLogFilename);
138
```

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

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

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

                links.Unsync.DisposeIfPossible();

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

            using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false, useLog: true))
            {
                var links = scope.Links;
                var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
                using (var transaction = transactionsLayer.BeginTransaction())
                {
                    l2 = links.Update(l2, l1);

                    links.Delete(l2);

                    ExceptionThrower();

                    transaction.Commit();
                }

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

            Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(lastScope.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
                ↪  UInt64ResizableDirectMemoryLinks(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
                ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                ↪  tempTransactionLogFilename))
259             using (var links = new UInt64Links(memoryAdapter))
260             {
261                 using (var transaction = memoryAdapter.BeginTransaction())
262                 {
263                     var l1 = links.CreateAndUpdate(itself, itself);
264                     var l2 = links.CreateAndUpdate(itself, itself);
265
266                     Global.Trash = links.Update(l2, l2, l1, l2);
267
268                     links.Delete(l1);
269
270                     transaction.Commit();
271                 }
272
273                 Global.Trash = links.Count();
274             }
275
276             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪  sactionLogFilename);
277
278             // Damage database
279
280             FileHelpers.WriteFirst(tempTransactionLogFilename, new
                ↪  UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
281
282             // Try load damaged database
283             try
284             {
285                 // TODO: Fix
```

```
286                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪  UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
                    ↪  tempTransactionLogFilename))
287                 using (var links = new UInt64Links(memoryAdapter))
288                 {
289                     Global.Trash = links.Count();
290                 }
291             }
292             catch (NotSupportedException ex)
293             {
294                 Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
                    ↪  yet.");
295             }
296 
297             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪  sactionLogFilename);
298 
299             File.Delete(tempDatabaseFilename);
300             File.Delete(tempTransactionLogFilename);
301         }
302 
303         [Fact]
304         public static void Bug1Test()
305         {
306             var tempDatabaseFilename = Path.GetTempFileName();
307             var tempTransactionLogFilename = Path.GetTempFileName();
308 
309             var itself = _constants.Itself;
310 
311             // User Code Error (Autoreverted), some data saved
312             try
313             {
314                 ulong l1;
315                 ulong l2;
316 
317                 using (var memory = new UInt64ResizableDirectMemoryLinks(tempDatabaseFilename))
318                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
                    ↪  tempTransactionLogFilename))
319                 using (var links = new UInt64Links(memoryAdapter))
320                 {
321                     l1 = links.CreateAndUpdate(itself, itself);
322                     l2 = links.CreateAndUpdate(itself, itself);
323 
324                     l2 = links.Update(l2, l2, l1, l2);
325 
326                     links.CreateAndUpdate(l2, itself);
327                     links.CreateAndUpdate(l2, itself);
328                 }
329 
330                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                    ↪  TransactionLogFilename);
331 
332                 using (var memory = new UInt64ResizableDirectMemoryLinks(tempDatabaseFilename))
333                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
                    ↪  tempTransactionLogFilename))
334                 using (var links = new UInt64Links(memoryAdapter))
335                 {
336                     using (var transaction = memoryAdapter.BeginTransaction())
337                     {
338                         l2 = links.Update(l2, l1);
339 
340                         links.Delete(l2);
341 
342                         ExceptionThrower();
343 
344                         transaction.Commit();
345                     }
346 
347                     Global.Trash = links.Count();
348                 }
349             }
350             catch
351             {
352                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                    ↪  TransactionLogFilename);
353             }
354 
355             File.Delete(tempDatabaseFilename);
356             File.Delete(tempTransactionLogFilename);
```

```
357         }
358
359         private static void ExceptionThrower() => throw new InvalidOperationException();
360
361         [Fact]
362         public static void PathsTest()
363         {
364             var source = _constants.SourcePart;
365             var target = _constants.TargetPart;
366
367             using (var scope = new TempLinksTestScope())
368             {
369                 var links = scope.Links;
370                 var l1 = links.CreatePoint();
371                 var l2 = links.CreatePoint();
372
373                 var r1 = links.GetByKeys(l1, source, target, source);
374                 var r2 = links.CheckPathExistance(l2, l2, l2, l2);
375             }
376         }
377
378         [Fact]
379         public static void RecursiveStringFormattingTest()
380         {
381             using (var scope = new TempLinksTestScope(useSequences: true))
382             {
383                 var links = scope.Links;
384                 var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
385
386                 var a = links.CreatePoint();
387                 var b = links.CreatePoint();
388                 var c = links.CreatePoint();
389
390                 var ab = links.GetOrCreate(a, b);
391                 var cb = links.GetOrCreate(c, b);
392                 var ac = links.GetOrCreate(a, c);
393
394                 a = links.Update(a, c, b);
395                 b = links.Update(b, a, c);
396                 c = links.Update(c, a, b);
397
398                 Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
399                 Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
400                 Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
401
402                 Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
                    ↪  "(5:(4:5 (6:5 4)) 6)");
403                 Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
                    ↪  "(6:(5:(4:5 6) 6) 4)");
404                 Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
                    ↪  "(4:(5:4 (6:5 4)) 6)");
405
406                 // TODO: Think how to build balanced syntax tree while formatting structure (eg.
                    ↪  "(4:(5:4 6) (6:5 4)") instead of "(4:(5:4 (6:5 4)) 6)"
407
408                 Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
                    ↪  "{{5}{5}{4}{6}}");
409                 Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
                    ↪  "{{5}{6}{6}{4}}");
410                 Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
                    ↪  "{{4}{5}{4}{6}}");
411             }
412         }
413
414         private static void DefaultFormatter(StringBuilder sb, ulong link)
415         {
416             sb.Append(link.ToString());
417         }
418
419         #endregion
420
421         #region Performance
422
423         /*
424         public static void RunAllPerformanceTests()
425         {
426             try
427             {
428                 links.TestLinksInSteps();
```

```
            }
            catch (Exception ex)
            {
                ex.WriteToConsole();
            }

            return;

            try
            {
                //ThreadPool.SetMaxThreads(2, 2);

                // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
↪  результат
                // Также это дополнительно помогает в отладке
                // Увеличивает вероятность попадания информации в кэши
                for (var i = 0; i < 10; i++)
                {
                    //0 - 10 ГБ
                    //Каждые 100 МБ срез цифр

                    //links.TestGetSourceFunction();
                    //links.TestGetSourceFunctionInParallel();
                    //links.TestGetTargetFunction();
                    //links.TestGetTargetFunctionInParallel();
                    links.Create64BillionLinks();

                    links.TestRandomSearchFixed();
                    //links.Create64BillionLinksInParallel();
                    links.TestEachFunction();
                    //links.TestForeach();
                    //links.TestParallelForeach();
                }

                links.TestDeletionOfAllLinks();

            }
            catch (Exception ex)
            {
                ex.WriteToConsole();
            }
        }*/

         /*
        public static void TestLinksInSteps()
        {
            const long gibibyte = 1024 * 1024 * 1024;
            const long mebibyte = 1024 * 1024;

            var totalLinksToCreate = gibibyte /
↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
            var linksStep = 102 * mebibyte /
↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;

            var creationMeasurements = new List<TimeSpan>();
            var searchMeasuremets = new List<TimeSpan>();
            var deletionMeasurements = new List<TimeSpan>();

            GetBaseRandomLoopOverhead(linksStep);
            GetBaseRandomLoopOverhead(linksStep);

            var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);

            ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);

            var loops = totalLinksToCreate / linksStep;

            for (int i = 0; i < loops; i++)
            {
                creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
                searchMeasuremets.Add(Measure(() => links.RunRandomSearches(linksStep)));

                Console.Write("\rC + S {0}/{1}", i + 1, loops);
            }

            ConsoleHelpers.Debug();

            for (int i = 0; i < loops; i++)
            {
                deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
```

```
506
507                 Console.Write("\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],
    ↪  searchMeasuremets[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,
    ↪  searchMeasuremets[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
524             }
525
526             ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
    ↪  links.Total);
527         }
528
529         private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
    ↪  amountToCreate)
530         {
531             for (long i = 0; i < amountToCreate; i++)
532                 links.Create(0, 0);
533         }
534
535          private static TimeSpan GetBaseRandomLoopOverhead(long loops)
536          {
537              return Measure(() =>
538              {
539                  ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
540                  ulong result = 0;
541                  for (long i = 0; i < loops; i++)
542                  {
543                      var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
544                      var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
545
546                      result += maxValue + source + target;
547                  }
548                  Global.Trash = result;
549              });
550          }
551           */
552
553          [Fact(Skip = "performance test")]
554          public static void GetSourceTest()
555          {
556              using (var scope = new TempLinksTestScope())
557              {
558                  var links = scope.Links;
559                  ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
                     ↪  Iterations);
560
561                  ulong counter = 0;
562
563                  //var firstLink = links.First();
564                  // Создаём одну связь, из которой будет производить считывание
565                  var firstLink = links.Create();
566
567                  var sw = Stopwatch.StartNew();
568
569                  // Тестируем саму функцию
570                  for (ulong i = 0; i < Iterations; i++)
571                  {
572                      counter += links.GetSource(firstLink);
573                  }
574
575                  var elapsedTime = sw.Elapsed;
576
577                  var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
578
579                  // Удаляем связь, из которой производилось считывание
```

```
580                 links.Delete(firstLink);
581
582                 ConsoleHelpers.Debug(
583                     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
584                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
585             }
586         }
587
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
                    ↪  parallel.", Iterations);
595
596                 long counter = 0;
597
598                 //var firstLink = links.First();
599                 var firstLink = links.Create();
600
601                 var sw = Stopwatch.StartNew();
602
603                 // Тестируем саму функцию
604                 Parallel.For(0, Iterations, x =>
605                 {
606                     Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
607                     //Interlocked.Increment(ref counter);
608                 });
609
610                 var elapsedTime = sw.Elapsed;
611
612                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
613
614                 links.Delete(firstLink);
615
616                 ConsoleHelpers.Debug(
617                     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
618                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
619             }
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.",
                    ↪  Iterations);
629
630                 ulong counter = 0;
631
632                 //var firstLink = links.First();
633                 var firstLink = links.Create();
634
635                 var sw = Stopwatch.StartNew();
636
637                 for (ulong i = 0; i < Iterations; i++)
638                 {
639                     counter += links.GetTarget(firstLink);
640                 }
641
642                 var elapsedTime = sw.Elapsed;
643
644                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
645
646                 links.Delete(firstLink);
647
648                 ConsoleHelpers.Debug(
649                     "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
650                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
651             }
652         }
653
```

```
654         [Fact(Skip = "performance test")]
655         public static void TestGetTargetInParallel()
656         {
657             using (var scope = new TempLinksTestScope())
658             {
659                 var links = scope.Links;
660                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
                    ↪  parallel.", Iterations);
661 
662                 long counter = 0;
663 
664                 //var firstLink = links.First();
665                 var firstLink = links.Create();
666 
667                 var sw = Stopwatch.StartNew();
668 
669                 Parallel.For(0, Iterations, x =>
670                 {
671                     Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
672                     //Interlocked.Increment(ref counter);
673                 });
674 
675                 var elapsedTime = sw.Elapsed;
676 
677                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
678 
679                 links.Delete(firstLink);
680 
681                 ConsoleHelpers.Debug(
682                     "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
683                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
684             }
685         }
686 
687         // TODO: Заполнить базу данных перед тестом
688         /*
689         [Fact]
690         public void TestRandomSearchFixed()
691         {
692             var tempFilename = Path.GetTempFileName();
693 
694             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
695             {
696                 long iterations = 64 * 1024 * 1024 /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
697 
698                 ulong counter = 0;
699                 var maxLink = links.Total;
700 
701                 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
702 
703                 var sw = Stopwatch.StartNew();
704 
705                 for (var i = iterations; i > 0; i--)
706                 {
707                     var source =
    ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
708                     var target =
    ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
709 
710                     counter += links.Search(source, target);
711                 }
712 
713                 var elapsedTime = sw.Elapsed;
714 
715                 var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
716 
717                 ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
    ↪  Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
    ↪  counter);
718             }
719 
720             File.Delete(tempFilename);
721         }*/
722 
723         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
724         public static void TestRandomSearchAll()
```

```
725         {
726             using (var scope = new TempLinksTestScope())
727             {
728                 var links = scope.Links;
729                 ulong counter = 0;
730 
731                 var maxLink = links.Count();
732 
733                 var iterations = links.Count();
734 
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}",
754                      iterations, elapsedTime, (long)iterationsPerSecond, counter);
755             }
756         }
757 
758         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
759         public static void TestEach()
760         {
761             using (var scope = new TempLinksTestScope())
762             {
763                 var links = scope.Links;
764 
765                 var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
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)",
778                     counter, elapsedTime, (long)linksPerSecond);
779             }
780         }
781 
782         /*
783         [Fact]
784         public static void TestForeach()
785         {
786             var tempFilename = Path.GetTempFileName();
787 
788             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
789             {
790                 ulong counter = 0;
791 
792                 ConsoleHelpers.Debug("Testing foreach through links.");
793 
794                 var sw = Stopwatch.StartNew();
795 
796                 //foreach (var link in links)
797                 //{
798                 //    counter++;
799                 //}
```

```
800
801                 var elapsedTime = sw.Elapsed;
802
803                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
804
805                 ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
    ↪  links per second)", counter, elapsedTime, (long)linksPerSecond);
806             }
807
808             File.Delete(tempFilename);
809         }
810         */
811
812         /*
813         [Fact]
814         public static void TestParallelForeach()
815         {
816             var tempFilename = Path.GetTempFileName();
817
818             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
819             {
820
821                 long counter = 0;
822
823                 ConsoleHelpers.Debug("Testing parallel foreach through links.");
824
825                 var sw = Stopwatch.StartNew();
826
827                 //Parallel.ForEach((IEnumerable<ulong>)links, x =>
828                 //{
829                 //    Interlocked.Increment(ref counter);
830                 //});
831
832                 var elapsedTime = sw.Elapsed;
833
834                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
835
836                 ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
    ↪  {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
837             }
838
839             File.Delete(tempFilename);
840         }
841         */
842
843         [Fact(Skip = "performance test")]
844         public static void Create64BillionLinks()
845         {
846             using (var scope = new TempLinksTestScope())
847             {
848                 var links = scope.Links;
849                 var linksBeforeTest = links.Count();
850
851                 long linksToCreate = 64 * 1024 * 1024 /
                     ↪  UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
852
853                 ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
854
855                 var elapsedTime = Performance.Measure(() =>
856                 {
857                     for (long i = 0; i < linksToCreate; i++)
858                     {
859                         links.Create();
860                     }
861                 });
862
863                 var linksCreated = links.Count() - linksBeforeTest;
864                 var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
865
866                 ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
867
868                 ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                     ↪  linksCreated, elapsedTime,
869                     (long)linksPerSecond);
870             }
871         }
872
873         [Fact(Skip = "performance test")]
```

```
874         public static void Create64BillionLinksInParallel()
875         {
876             using (var scope = new TempLinksTestScope())
877             {
878                 var links = scope.Links;
879                 var linksBeforeTest = links.Count();
880 
881                 var sw = Stopwatch.StartNew();
882 
883                 long linksToCreate = 64 * 1024 * 1024 /
                    ↪  UInt64ResizableDirectMemoryLinks.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)",
                    ↪  linksCreated, elapsedTime,
895                     (long)linksPerSecond);
896             }
897         }
898 
899         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
900         public static void TestDeletionOfAllLinks()
901         {
902             using (var scope = new TempLinksTestScope())
903             {
904                 var links = scope.Links;
905                 var linksBeforeTest = links.Count();
906 
907                 ConsoleHelpers.Debug("Deleting all links");
908 
909                 var elapsedTime = Performance.Measure(links.DeleteAll);
910 
911                 var linksDeleted = linksBeforeTest - links.Count();
912                 var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
913 
914                 ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
                    ↪  linksDeleted, elapsedTime,
915                     (long)linksPerSecond);
916             }
917         }
918 
919         #endregion
920     }
921 }
```

1.114 ./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
                    ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
19                 var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
                    ↪  powerOf2ToUnaryNumberConverter);
20                 var random = new System.Random(0);
21                 ulong[] numbers = new ulong[N];
22                 ulong[] unaryNumbers = new ulong[N];
23                 for (int i = 0; i < N; i++)
24                 {
25                     numbers[i] = random.NextUInt64();
```

```
                    unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
                }
                var fromUnaryNumberConverterUsingOrOperation = new
                ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                ↪  powerOf2ToUnaryNumberConverter);
                var fromUnaryNumberConverterUsingAddOperation = new
                ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
                for (int i = 0; i < N; i++)
                {
                    Assert.Equal(numbers[i],
                    ↪  fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
                    Assert.Equal(numbers[i],
                    ↪  fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
                }
            }
        }
    }
}
```

1.115 ./Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs

```
using Xunit;
using Platform.Converters;
using Platform.Memory;
using Platform.Reflection;
using Platform.Scopes;
using Platform.Data.Numbers.Raw;
using Platform.Data.Doublets.Incrementers;
using Platform.Data.Doublets.Numbers.Unary;
using Platform.Data.Doublets.PropertyOperators;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Unicode;
using Platform.Data.Doublets.ResizableDirectMemory.Generic;

namespace Platform.Data.Doublets.Tests
{
    public static class UnicodeConvertersTests
    {
        [Fact]
        public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var meaningRoot = links.CreatePoint();
                var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
                var powerOf2ToUnaryNumberConverter = new
                ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
                var addressToUnaryNumberConverter = new
                ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
                var unaryNumberToAddressConverter = new
                ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                ↪  powerOf2ToUnaryNumberConverter);
                TestCharAndUnicodeSymbolConverters(links, meaningRoot,
                ↪  addressToUnaryNumberConverter, unaryNumberToAddressConverter);
            }
        }

        [Fact]
        public static void CharAndRawNumberUnicodeSymbolConvertersTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  ResizableDirectMemoryLinks<ulong>>>())
            {
                var links = scope.Use<ILinks<ulong>>();
                var meaningRoot = links.CreatePoint();
                var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
                var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
                TestCharAndUnicodeSymbolConverters(links, meaningRoot,
                ↪  addressToRawNumberConverter, rawNumberToAddressConverter);
            }
        }

        private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
        ↪  meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
        ↪  numberToAddressConverter)
        {
            var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
```

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

```
                    var unicodeSequenceToStringConverter = new
                    ↪  UnicodeSequenceToStringConverter<ulong>(links,
                    ↪  unicodeSequenceCriterionMatcher, sequenceWalker,
                    ↪  unicodeSymbolToCharConverter);

                    var resultingString =
                    ↪  unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);

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

Index