LinksPlatform's Platform.Data.Doublets.Lino Class Library

1.1 ./csharp/Platform.Data.Doublets.Lino/DefaultLinoStorage.cs

```
using System.Collections.Generic;
using Platform.Converters;
using LinoLink = Platform.Communication.Protocol.Lino.Link;

namespace Platform.Data.Doublets.Lino;

public class DefaultLinoStorage<TLinkAddress> : ILinoStorage<TLinkAddress>
{
    private readonly ILinks<TLinkAddress> _storage;

    public DefaultLinoStorage(ILinks<TLinkAddress> storage)
    {
        _storage = storage;
    }

    public void CreateLinks(IList<LinoLink> links)
    {
        var checkedConverter = CheckedConverter<ulong, TLinkAddress>.Default;
        for (int i = 0; i < links.Count; i++)
        {
            _storage.Create();
        }
        for (int i = 0; i < links.Count; i++)
        {
            var index = ulong.Parse(links[i].Id);
            var source = ulong.Parse(links[i].Values[0].Id);
            var target = ulong.Parse(links[i].Values[1].Id);
            _storage.Update(checkedConverter.Convert(index), checkedConverter.Convert(source),
            ↪  checkedConverter.Convert(target));
        }
    }

    public IList<LinoLink> GetLinks()
    {
        var allLinks = _storage.All();
        var linoLinks = new List<LinoLink>(allLinks.Count);
        for (int i = 0; i < allLinks.Count; i++)
        {
            var link = new Link<TLinkAddress>(allLinks[i]);
            var linoLink = new LinoLink(link.Index.ToString(), new List<LinoLink> {
            ↪  link.Source.ToString(), link.Target.ToString() });
            linoLinks.Add(linoLink);
        }
        return linoLinks;
    }
}
```

1.2 ./csharp/Platform.Data.Doublets.Lino/ILinoStorage.cs

```
using System.Collections.Generic;
using Platform.Communication.Protocol.Lino;

namespace Platform.Data.Doublets.Lino;

public interface ILinoStorage<TLinkAddress>
{
    void CreateLinks(IList<Link> links);
    IList<Link> GetLinks();
}
```

1.3 ./csharp/Platform.Data.Doublets.Lino/LinoDocumentsStorage.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using Platform.Collections.Stacks;
using Platform.Converters;
using Platform.Data.Doublets.CriterionMatchers;
using Platform.Data.Doublets.Numbers.Rational;
using Platform.Data.Doublets.Numbers.Raw;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Unicode;
using Platform.Data.Numbers.Raw;
using Platform.Numbers;
using LinoLink = Platform.Communication.Protocol.Lino.Link;

namespace Platform.Data.Doublets.Lino;

public class LinoDocumentsStorage<TLinkAddress> : ILinoStorage<TLinkAddress>
```

```
{
    private static readonly TLinkAddress Zero = default!;
    private static readonly TLinkAddress One = Arithmetic.Increment(Zero);

    private readonly EqualityComparer<TLinkAddress> _equalityComparer =
    ↪  EqualityComparer<TLinkAddress>.Default;

    // Converters that are able to convert link's address (UInt64 value) to a raw number
    ↪  represented with another UInt64 value and back
    private readonly RawNumberToAddressConverter<TLinkAddress> _numberToAddressConverter = new();
    private readonly AddressToRawNumberConverter<TLinkAddress> _addressToNumberConverter = new();

    private ILinks<TLinkAddress> Storage { get; }

    private IConverter<string, TLinkAddress> StringToUnicodeSequenceConverter { get; }

    private IConverter<TLinkAddress, string> UnicodeSequenceToStringConverter { get; }

    public TLinkAddress DocumentMarker { get; }

    public TLinkAddress ReferenceMarker { get; }

    public TLinkAddress LinkWithoutIdMarker { get; }

    public TLinkAddress LinkWithIdMarker { get; }

    private readonly IConverter<IList<TLinkAddress>?, TLinkAddress> _listToSequenceConverter;

    public LinoDocumentsStorage(ILinks<TLinkAddress> storage, IConverter<IList<TLinkAddress>?,
    ↪  TLinkAddress> listToSequenceConverter)
    {
        Storage = storage;
        // Initializes constants
        var markerIndex = One;
        var meaningRoot = storage.GetOrCreate(markerIndex, markerIndex);
        var unicodeSymbolMarker = storage.GetOrCreate(meaningRoot, Arithmetic.Increment(ref
        ↪  markerIndex));
        var unicodeSequenceMarker = storage.GetOrCreate(meaningRoot, Arithmetic.Increment(ref
        ↪  markerIndex));
        DocumentMarker = storage.GetOrCreate(meaningRoot, Arithmetic.Increment(ref markerIndex));
        ReferenceMarker = storage.GetOrCreate(meaningRoot, Arithmetic.Increment(ref
        ↪  markerIndex));
        LinkWithoutIdMarker = storage.GetOrCreate(meaningRoot, Arithmetic.Increment(ref
        ↪  markerIndex));
        LinkWithIdMarker = storage.GetOrCreate(meaningRoot, Arithmetic.Increment(ref
        ↪  markerIndex));
        BalancedVariantConverter<TLinkAddress> balancedVariantConverter = new(storage);
        TargetMatcher<TLinkAddress> unicodeSymbolCriterionMatcher = new(storage,
        ↪  unicodeSymbolMarker);
        TargetMatcher<TLinkAddress> unicodeSequenceCriterionMatcher = new(storage,
        ↪  unicodeSequenceMarker);
        CharToUnicodeSymbolConverter<TLinkAddress> charToUnicodeSymbolConverter = new(storage,
        ↪  _addressToNumberConverter, unicodeSymbolMarker);
        UnicodeSymbolToCharConverter<TLinkAddress> unicodeSymbolToCharConverter = new(storage,
        ↪  _numberToAddressConverter, unicodeSymbolCriterionMatcher);
        StringToUnicodeSequenceConverter = new CachingConverterDecorator<string,
        ↪  TLinkAddress>(new StringToUnicodeSequenceConverter<TLinkAddress>(storage,
        ↪  charToUnicodeSymbolConverter, balancedVariantConverter, unicodeSequenceMarker));
        RightSequenceWalker<TLinkAddress> sequenceWalker = new(storage, new
        ↪  DefaultStack<TLinkAddress>(), unicodeSymbolCriterionMatcher.IsMatched);
        UnicodeSequenceToStringConverter = new CachingConverterDecorator<TLinkAddress,
        ↪  string>(new UnicodeSequenceToStringConverter<TLinkAddress>(storage,
        ↪  unicodeSequenceCriterionMatcher, sequenceWalker, unicodeSymbolToCharConverter));
        _listToSequenceConverter = listToSequenceConverter;
    }

    private bool IsLinkWithId(TLinkAddress linkAddress) =>
    ↪  IsLinkWithId(Storage.GetLink(linkAddress));

    private bool IsLinkWithoutId(TLinkAddress linkAddress) =>
    ↪  IsLinkWithoutId(Storage.GetLink(linkAddress));

    private bool IsLink(TLinkAddress linkAddress)
    {
        var link = Storage.GetLink(linkAddress);
        return IsLinkWithId(link) || IsLinkWithoutId(link);
    }

```

```
78      private bool IsLinkWithId(IList<TLinkAddress> link)
79      {
80          var source = Storage.GetSource(link);
81          return _equalityComparer.Equals(LinkWithIdMarker, source);
82      }
83
84      private bool IsLinkWithoutId(IList<TLinkAddress> link)
85      {
86          var source = Storage.GetSource(link);
87          return _equalityComparer.Equals(LinkWithoutIdMarker, source);
88      }
89
90      private bool IsLink(IList<TLinkAddress> link) => IsLinkWithId(link) || IsLinkWithoutId(link);
91
92      private bool IsReference(TLinkAddress reference)
93      {
94          var link = Storage.GetLink(reference);
95          return IsReference(link);
96      }
97
98      private bool IsReference(IList<TLinkAddress> reference)
99      {
100         var source = Storage.GetSource(reference);
101         return _equalityComparer.Equals(ReferenceMarker, source);
102     }
103
104
105     private TLinkAddress GetOrCreateReference(string content)
106     {
107         var sequence = StringToUnicodeSequenceConverter.Convert(content);
108         return Storage.GetOrCreate(ReferenceMarker, sequence);
109     }
110
111     private string ReadReference(TLinkAddress reference) =>
        ↪ ReadReference(Storage.GetLink(reference));
112
113     private string ReadReference(IList<TLinkAddress> reference)
114     {
115         var referenceLink = new Link<TLinkAddress>(reference);
116         if (!_equalityComparer.Equals(ReferenceMarker, referenceLink.Source))
117         {
118             throw new ArgumentException("The passed link is not a reference");
119         }
120         return UnicodeSequenceToStringConverter.Convert(referenceLink.Target);
121     }
122
123     public void CreateLinks(IList<LinoLink> links)
124     {
125         var sequenceList = new List<TLinkAddress>();
126         for (int i = 0; i < links.Count; i++)
127         {
128             sequenceList.Add(CreateLink(links[i]));
129         }
130         Storage.GetOrCreate(DocumentMarker, _listToSequenceConverter.Convert(sequenceList));
131     }
132
133     private TLinkAddress CreateLink(LinoLink link)
134     {
135         if (link.Id != null)
136         {
137             return CreateLinkWithId(link);
138         }
139         var valuesSequence = CreateValuesSequence(link);
140         return Storage.GetOrCreate(LinkWithoutIdMarker, valuesSequence);
141     }
142
143     private TLinkAddress CreateLinkWithId(LinoLink link)
144     {
145         var currentReference = GetOrCreateReference(link.Id);
146         if (link.Values == null)
147         {
148             return Storage.GetOrCreate(LinkWithIdMarker, currentReference);
149         }
150         var valuesSequence = CreateValuesSequence(link);
151         var idWithValues = Storage.GetOrCreate(currentReference, valuesSequence);
152         return Storage.GetOrCreate(LinkWithIdMarker, idWithValues);
153     }
154
155     private TLinkAddress CreateValuesSequence(LinoLink parent)
```

```
156     {
157         var values = new List<TLinkAddress>(parent.Values.Count);
158         for (int i = 0; i < parent.Values.Count; i++)
159         {
160             var currentValue = parent.Values[i];
161             if (currentValue.Values != null)
162             {
163                 var valueLink = CreateLink(currentValue);
164                 values.Add(valueLink);
165                 continue;
166             }
167             var currentValueReference = GetOrCreateReference(currentValue.Id);
168             values.Add(currentValueReference);
169         }
170         return _listToSequenceConverter.Convert(values);
171     }
172 
173     private TLinkAddress GetDocumentSequence()
174     {
175         var constants = Storage.Constants;
176         var any = constants.Any;
177         TLinkAddress documentLinksSequence = default;
178         Storage.Each(new Link<TLinkAddress>(any, DocumentMarker, any), link =>
179         {
180             documentLinksSequence = Storage.GetTarget((IList<TLinkAddress>)link);
181             return constants.Continue;
182         });
183         if (_equalityComparer.Equals(default, documentLinksSequence))
184         {
185             throw new Exception("No document links in the storage.");
186         }
187         return documentLinksSequence;
188     }
189 
190     public IList<LinoLink> GetLinks()
191     {
192         var resultLinks = new List<LinoLink>();
193         bool IsElement(TLinkAddress linkIndex)
194         {
195             var source = Storage.GetSource(linkIndex);
196             return _equalityComparer.Equals(LinkWithoutIdMarker, source) |
                ↪  _equalityComparer.Equals(LinkWithIdMarker, source) ||
                ↪  Storage.IsPartialPoint(linkIndex);
197         }
198         var rightSequenceWalker = new RightSequenceWalker<TLinkAddress>(Storage, new
            ↪  DefaultStack<TLinkAddress>(), IsElement);
199         var documentLinksSequence = GetDocumentSequence();
200         var documentLinks = rightSequenceWalker.Walk(documentLinksSequence);
201         foreach (var documentLink in documentLinks)
202         {
203             resultLinks.Add(GetLink(documentLink));
204         }
205         return resultLinks;
206     }
207 
208     private bool IsLinkOrReferenceOrPartialPoint(TLinkAddress linkIndex)
209     {
210         var link = Storage.GetLink(linkIndex);
211         return IsLink(link) || IsReference(link) || Storage.IsPartialPoint(linkIndex);
212     }
213 
214     private LinoLink GetLinkWithId(IList<TLinkAddress> linkWithId)
215     {
216         string id;
217         var values = new List<LinoLink>();
218         var linkStruct = new Link<TLinkAddress>(linkWithId);
219         if (IsReference(linkStruct.Target))
220         {
221             id = ReadReference(linkStruct.Target);
222             return new LinoLink(id);
223         }
224         var rightSequenceWalker = new RightSequenceWalker<TLinkAddress>(Storage, new
            ↪  DefaultStack<TLinkAddress>(), IsLinkOrReferenceOrPartialPoint);
225         using var enumerator = rightSequenceWalker.Walk(linkStruct.Target).GetEnumerator();
226         enumerator.MoveNext();
227         id = ReadReference(enumerator.Current);
228         while (enumerator.MoveNext())
229         {
```

```
            var currentValueStruct = new Link<TLinkAddress>(Storage.GetLink(enumerator.Current));
            if (IsLink(currentValueStruct))
            {
                var value = GetLink(currentValueStruct);
                values.Add(value);
            }
            else if (IsReference(currentValueStruct))
            {
                var reference = ReadReference(currentValueStruct);
                var currentLinoLink = new LinoLink(reference);
                values.Add(currentLinoLink);
            }
        }
        return new LinoLink(id, values);
    }

    public LinoLink GetLinkWithoutId(IList<TLinkAddress> linkWithoudId)
    {
        var values = new List<LinoLink>();
        var linkStruct = new Link<TLinkAddress>(linkWithoudId);
        var rightSequenceWalker = new RightSequenceWalker<TLinkAddress>(Storage, new
        ↪  DefaultStack<TLinkAddress>(), IsLinkOrReferenceOrPartialPoint);
        foreach (var currentValue in rightSequenceWalker.Walk(linkStruct.Target))
        {
            var currentValueStruct = new Link<TLinkAddress>(Storage.GetLink(currentValue));
            if (IsLink(currentValue))
            {
                var value = GetLink(currentValueStruct);
                values.Add(value);
            }
            else if (IsReference(currentValueStruct))
            {
                var currentLinoLink = new LinoLink(ReadReference(currentValueStruct));
                values.Add(currentLinoLink);
            }
        }
        return new LinoLink(null, values);
    }

    public LinoLink GetLink(TLinkAddress link) => GetLink(Storage.GetLink(link));

    public LinoLink GetLink(IList<TLinkAddress> link)
    {
        var linkStruct = new Link<TLinkAddress>(link);
        if (IsLinkWithId(linkStruct))
        {
            return GetLinkWithId(linkStruct);
        }
        if (IsLinkWithoutId(linkStruct))
        {
            return GetLinkWithoutId(linkStruct);
        }
        throw new Exception("The passed argument is not a link");
    }
}
```

1.4 ./csharp/Platform.Data.Doublets.Lino/LinoExporter.cs

```
using System.Collections.Generic;
using Platform.Communication.Protocol.Lino;

namespace Platform.Data.Doublets.Lino;

public class LinoExporter<TLinkAddress>
{
    private readonly EqualityComparer<TLinkAddress> _equalityComparer =
    ↪  EqualityComparer<TLinkAddress>.Default;
    private readonly ILinoStorage<TLinkAddress> _linoDocumentsStorage;

    public LinoExporter(ILinoStorage<TLinkAddress> linoDocumentsStorage)
    {
        _linoDocumentsStorage = linoDocumentsStorage;
    }

    public string GetAllLinks()
    {
        var allLinks = _linoDocumentsStorage.GetLinks();
        return allLinks.Format();
    }
}
```

1.5 ./csharp/Platform.Data.Doublets.Lino/LinoExporterCli.cs

```
using System;
using System.IO;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.IO;
using Platform.Memory;

namespace Platform.Data.Doublets.Lino;

public class LinoExporterCli<TLinkAddress> where TLinkAddress : struct
{
    public void Run(params string[] args)
    {
        var argumentsIndex = 0;
        var storageFilePath = ConsoleHelpers.GetOrReadArgument(argumentsIndex++, "A path to a links storage", args);
        var notationFilePath = ConsoleHelpers.GetOrReadArgument(argumentsIndex++, "A path to a notation file", args);
        var documentName = ConsoleHelpers.GetOrReadArgument(argumentsIndex++, "A document name", args);
        using var linksMemory = new FileMappedResizableDirectMemory(storageFilePath);
        var storage = new UnitedMemoryLinks<TLinkAddress>(linksMemory).DecorateWithAutomaticUniquenessAndUsagesResolution();
        ILinoStorage<TLinkAddress> linoStorage;
        if (String.IsNullOrWhiteSpace(documentName))
        {
            linoStorage = new DefaultLinoStorage<TLinkAddress>(storage);
        }
        else
        {
            linoStorage = new LinoDocumentsStorage<TLinkAddress>(storage, new BalancedVariantConverter<TLinkAddress>(storage));
        }
        var exporter = new LinoExporter<TLinkAddress>(linoStorage);
        var allLinksNotation = exporter.GetAllLinks();
        File.WriteAllText(notationFilePath, allLinksNotation);
    }
}
```

1.6 ./csharp/Platform.Data.Doublets.Lino/LinoImporter.cs

```
using Platform.Communication.Protocol.Lino;

namespace Platform.Data.Doublets.Lino;

public class LinoImporter<TLinkAddress>
{
    private readonly ILinoStorage<TLinkAddress> _linoDocumentsStorage;
    private readonly Parser _parser = new Parser();

    public LinoImporter(ILinoStorage<TLinkAddress> linoDocumentsStorage)
    {
        _linoDocumentsStorage = linoDocumentsStorage;
    }

    public void Import(string content)
    {
        var linoLinks = _parser.Parse(content);
        _linoDocumentsStorage.CreateLinks(linoLinks);
    }

}
```

1.7 ./csharp/Platform.Data.Doublets.Lino/LinoImporterCli.cs

```
using System;
using System.IO;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.IO;
using Platform.Memory;

namespace Platform.Data.Doublets.Lino;

public class LinoImporterCli<TLinkAddress> where TLinkAddress : struct
{
    public void Run(params string[] args)
    {
        var argumentsIndex = 0;
        var notationFilePath = ConsoleHelpers.GetOrReadArgument(argumentsIndex++, "A path to a notation file", args);
```

```
16          var storageFilePath = ConsoleHelpers.GetOrReadArgument(argumentsIndex++, "A path to a
            ↪  links storage", args);
17          var documentName = ConsoleHelpers.GetOrReadArgument(argumentsIndex++, "A document name",
            ↪  args);
18          using var linksMemory = new FileMappedResizableDirectMemory(storageFilePath);
19          var storage = new UnitedMemoryLinks<TLinkAddress>(linksMemory).DecorateWithAutomaticUniq
            ↪  uenessAndUsagesResolution();
20          ILinoStorage<TLinkAddress> linoStorage;
21          if (String.IsNullOrWhiteSpace(documentName))
22          {
23              linoStorage = new DefaultLinoStorage<TLinkAddress>(storage);
24          }
25          else
26          {
27              linoStorage = new LinoDocumentsStorage<TLinkAddress>(storage, new
                ↪  BalancedVariantConverter<TLinkAddress>(storage));
28          }
29          var importer = new LinoImporter<TLinkAddress>(linoStorage);
30          var notation = File.ReadAllText(notationFilePath);
31          importer.Import(notation);
32      }
33  }
```

1.8 ./csharp/Platform.Data.Doublets.Lino.Tests/ImporterAndExporterCliTests.cs

```
1   using System;
2   using System.IO;
3   using Platform.IO;
4   using Xunit;
5   using TLinkAddress = System.UInt64;
6
7   namespace Platform.Data.Doublets.Lino.Tests;
8
9   public class ImporterAndExporterCliTests
10  {
11      [Theory]
12      [InlineData("(1: 1 1)")]
13      [InlineData("(1: 1 1)\n(2: 2 2)")]
14      [InlineData("(1: 2 2)")]
15      [InlineData("(1: 2 2)\n(2: 1 1)")]
16      public void DefaultLinoStorageTest(string notation)
17      {
18          var notationFilePath = TemporaryFiles.UseNew();
19          var linksStorageFilePath = TemporaryFiles.UseNew();
20          var exportedNotationFilePath = TemporaryFiles.UseNew();
21          File.WriteAllText(notationFilePath, notation);
22          new LinoImporterCli<TLinkAddress>().Run(notationFilePath, linksStorageFilePath, "");
23          new LinoExporterCli<TLinkAddress>().Run(linksStorageFilePath, exportedNotationFilePath,
            ↪  "");
24          Assert.Equal(notation, File.ReadAllText(exportedNotationFilePath));
25      }
26
27      [Theory]
28      [InlineData("(1: 1 1)")]
29      [InlineData("(1: 1 1)\n(2: 2 2)")]
30      [InlineData("(2: 2 2)")]
31      [InlineData("(1: 2 2)")]
32      [InlineData("(1: 2 2)\n(2: 1 1)")]
33      [InlineData("(1: 2 (3: 3 3))")]
34      [InlineData("(1: 2 (3: 3 3))\n(2: 1 1)")]
35      [InlineData("(son: lovesMama)")]
36      [InlineData("(papa: (lovesMama: loves mama))")]
37      [InlineData(@"(papa (lovesMama: loves mama))
38  (son lovesMama)
39  (daughter lovesMama)
40  (all (love mama))")]
41      public void LinoDocumentsStorageTest(string notation)
42      {
43          var notationFilePath = TemporaryFiles.UseNew();
44          var linksStorageFilePath = TemporaryFiles.UseNew();
45          var exportedNotationFilePath = TemporaryFiles.UseNew();
46          File.WriteAllText(notationFilePath, notation);
47          new LinoImporterCli<TLinkAddress>().Run(notationFilePath, linksStorageFilePath,
            ↪  notationFilePath);
48          new LinoExporterCli<TLinkAddress>().Run(linksStorageFilePath, exportedNotationFilePath,
            ↪  notationFilePath);
49          Assert.Equal(notation, File.ReadAllText(exportedNotationFilePath));
50      }
51  }
```

1.9 ./csharp/Platform.Data.Doublets.Lino.Tests/ImporterAndExporterTests.cs

```
using Platform.Data.Doublets.Memory;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Memory;
using Xunit;
using TLinkAddress = System.UInt64;

namespace Platform.Data.Doublets.Lino.Tests;

public class ImporterAndExporterTests
{
    public static ILinks<TLinkAddress> CreateLinks() => CreateLinks(new IO.TemporaryFile());

    public static ILinks<TLinkAddress> CreateLinks(string dbFilename)
    {
        var linksConstants = new LinksConstants<TLinkAddress>(true);
        return new UnitedMemoryLinks<TLinkAddress>(new HeapResizableDirectMemory(),
        ↪  UnitedMemoryLinks<TLinkAddress>.DefaultLinksSizeStep, linksConstants,
        ↪  IndexTreeType.Default);
    }

    [InlineData("(1: 1 1)")]
    [InlineData("(1: 1 1)\n(2: 2 2)")]
    [InlineData("(1: 2 2)")]
    [InlineData("(1: 2 2)\n(2: 1 1)")]
    [Theory]
    public void LinoStorageTest(string notation)
    {
        var storage = CreateLinks();
        var linoStorage = new DefaultLinoStorage<TLinkAddress>(storage);
        var importer = new LinoImporter<TLinkAddress>(linoStorage);
        importer.Import(notation);
        var exporter = new LinoExporter<TLinkAddress>(linoStorage);
        var exportedLinks = exporter.GetAllLinks();
        Assert.Equal(notation, exportedLinks);
    }

    [InlineData("(1: 1 1)")]
    [InlineData("(1: 1 1)\n(2: 2 2)")]
    [InlineData("(2: 2 2)")]
    [InlineData("(1: 2 2)")]
    [InlineData("(1: 2 2)\n(2: 1 1)")]
    [InlineData("(1: 2 (3: 3 3))")]
    [InlineData("(1: 2 (3: 3 3))\n(2: 1 1)")]
    [InlineData("(son: lovesMama)")]
    [InlineData("(papa: (lovesMama: loves mama))")]
    [InlineData(@"(papa (lovesMama: loves mama))
(son lovesMama)
(daughter lovesMama)
(all (love mama))")]
    [Theory]
    public void LinoDocumentStorageTest(string notation)
    {
        var storage = CreateLinks();
        var linoStorage = new LinoDocumentsStorage<TLinkAddress>(storage, new
        ↪  BalancedVariantConverter<ulong>(storage));
        var importer = new LinoImporter<TLinkAddress>(linoStorage);
        importer.Import(notation);
        var exporter = new LinoExporter<TLinkAddress>(linoStorage);
        var exportedLinks = exporter.GetAllLinks();
        Assert.Equal(notation, exportedLinks);
    }

}
```

Index