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, string? documentName)
    {
        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(string? documentName)
    {
        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, string? documentName);
    IList<Link> GetLinks(string? documentName);
}
```

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, string? documentName)
124     {
125         if (string.IsNullOrEmpty(documentName))
126         {
127             throw new Exception("No document name is passed.");
128         }
129         var sequenceList = new List<TLinkAddress>();
130         for (int i = 0; i < links.Count; i++)
131         {
132             sequenceList.Add(CreateLink(links[i]));
133         }
134         var documentNameSequence = StringToUnicodeSequenceConverter.Convert(documentName);
135         var document = Storage.SearchOrDefault(documentNameSequence, documentNameSequence);
136         if (!_equalityComparer.Equals(default, document))
137         {
138             throw new Exception($"The document with name {documentName} already exists.");
139         }
140         document = Storage.CreateAndUpdate(DocumentMarker, documentNameSequence);
141         Storage.GetOrCreate(document, _listToSequenceConverter.Convert(sequenceList));
142     }
143
144     private TLinkAddress CreateLink(LinoLink link)
145     {
146         if (link.Id != null)
147         {
148             return CreateLinkWithId(link);
149         }
150         var valuesSequence = CreateValuesSequence(link);
151         return Storage.GetOrCreate(LinkWithoutIdMarker, valuesSequence);
152     }
153
154     private TLinkAddress CreateLinkWithId(LinoLink link)
```

```
    {
        var currentReference = GetOrCreateReference(link.Id);
        if (link.Values == null)
        {
            return Storage.GetOrCreate(LinkWithIdMarker, currentReference);
        }
        var valuesSequence = CreateValuesSequence(link);
        var idWithValues = Storage.GetOrCreate(currentReference, valuesSequence);
        return Storage.GetOrCreate(LinkWithIdMarker, idWithValues);
    }

    private TLinkAddress CreateValuesSequence(LinoLink parent)
    {
        var values = new List<TLinkAddress>(parent.Values.Count);
        for (int i = 0; i < parent.Values.Count; i++)
        {
            var currentValue = parent.Values[i];
            if (currentValue.Values != null)
            {
                var valueLink = CreateLink(currentValue);
                values.Add(valueLink);
                continue;
            }
            var currentValueReference = GetOrCreateReference(currentValue.Id);
            values.Add(currentValueReference);
        }
        return _listToSequenceConverter.Convert(values);
    }

    private TLinkAddress GetDocument(string documentName)
    {
        var documentNameSequence = StringToUnicodeSequenceConverter.Convert(documentName);
        var document = Storage.SearchOrDefault(DocumentMarker, documentNameSequence);
        if (_equalityComparer.Equals(default, document))
        {
            throw new Exception($"No document in the storage with name {documentName}.");
        }
        return document;
    }

    private TLinkAddress GetDocumentSequence(string documentName)
    {
        var document = GetDocument(documentName);
        return GetDocumentSequence(document);
    }

    private TLinkAddress GetDocumentSequence(TLinkAddress document)
    {
        var any = Storage.Constants.Any;
        TLinkAddress documentLinksSequence = default;
        Storage.Each(new Link<TLinkAddress>(any, document, any), link =>
        {
            documentLinksSequence = Storage.GetTarget(link);
            return Storage.Constants.Break;
        });
        if (_equalityComparer.Equals(default, documentLinksSequence))
        {
            throw new Exception("No links assosiated with the passed document in the storage.");
        }
        return documentLinksSequence;
    }

    public IList<LinoLink> GetLinks(string? documentName)
    {
        if (string.IsNullOrEmpty(documentName))
        {
            throw new Exception("No document name is passed.");
        }
        var resultLinks = new List<LinoLink>();
        bool IsElement(TLinkAddress linkIndex)
        {
            var source = Storage.GetSource(linkIndex);
            return _equalityComparer.Equals(LinkWithoutIdMarker, source) |
            ↪  _equalityComparer.Equals(LinkWithIdMarker, source) ||
            ↪  Storage.IsPartialPoint(linkIndex);
        }
        var document = GetDocument(documentName);
```

```
230             var rightSequenceWalker = new RightSequenceWalker<TLinkAddress>(Storage, new
                ↪  DefaultStack<TLinkAddress>(), IsElement);
231             var documentLinksSequence = GetDocumentSequence(document);
232             var documentLinks = rightSequenceWalker.Walk(documentLinksSequence);
233             foreach (var documentLink in documentLinks)
234             {
235                 resultLinks.Add(GetLink(documentLink));
236             }
237             return resultLinks;
238         }
239
240
241         private bool IsLinkOrReferenceOrPartialPoint(TLinkAddress linkIndex)
242         {
243             var link = Storage.GetLink(linkIndex);
244             return IsLink(link) || IsReference(link) || Storage.IsPartialPoint(linkIndex);
245         }
246
247         private LinoLink GetLinkWithId(IList<TLinkAddress> linkWithId)
248         {
249             string id;
250             var values = new List<LinoLink>();
251             var linkStruct = new Link<TLinkAddress>(linkWithId);
252             if (IsReference(linkStruct.Target))
253             {
254                 id = ReadReference(linkStruct.Target);
255                 return new LinoLink(id);
256             }
257             var rightSequenceWalker = new RightSequenceWalker<TLinkAddress>(Storage, new
                ↪  DefaultStack<TLinkAddress>(), IsLinkOrReferenceOrPartialPoint);
258             using var enumerator = rightSequenceWalker.Walk(linkStruct.Target).GetEnumerator();
259             enumerator.MoveNext();
260             id = ReadReference(enumerator.Current);
261             while (enumerator.MoveNext())
262             {
263                 var currentValueStruct = new Link<TLinkAddress>(Storage.GetLink(enumerator.Current));
264                 if (IsLink(currentValueStruct))
265                 {
266                     var value = GetLink(currentValueStruct);
267                     values.Add(value);
268                 }
269                 else if (IsReference(currentValueStruct))
270                 {
271                     var reference = ReadReference(currentValueStruct);
272                     var currentLinoLink = new LinoLink(reference);
273                     values.Add(currentLinoLink);
274                 }
275             }
276             return new LinoLink(id, values);
277         }
278
279         private LinoLink GetLinkWithoutId(IList<TLinkAddress> linkWithoudId)
280         {
281             var values = new List<LinoLink>();
282             var linkStruct = new Link<TLinkAddress>(linkWithoudId);
283             var rightSequenceWalker = new RightSequenceWalker<TLinkAddress>(Storage, new
                ↪  DefaultStack<TLinkAddress>(), IsLinkOrReferenceOrPartialPoint);
284             foreach (var currentValue in rightSequenceWalker.Walk(linkStruct.Target))
285             {
286                 var currentValueStruct = new Link<TLinkAddress>(Storage.GetLink(currentValue));
287                 if (IsLink(currentValue))
288                 {
289                     var value = GetLink(currentValueStruct);
290                     values.Add(value);
291                 }
292                 else if (IsReference(currentValueStruct))
293                 {
294                     var currentLinoLink = new LinoLink(ReadReference(currentValueStruct));
295                     values.Add(currentLinoLink);
296                 }
297             }
298             return new LinoLink(null, values);
299         }
300
301         private LinoLink GetLink(TLinkAddress link) => GetLink(Storage.GetLink(link));
302
303         private LinoLink GetLink(IList<TLinkAddress> link)
304         {
```

```
305             var linkStruct = new Link<TLinkAddress>(link);
306             if (IsLinkWithId(linkStruct))
307             {
308                 return GetLinkWithId(linkStruct);
309             }
310             if (IsLinkWithoutId(linkStruct))
311             {
312                 return GetLinkWithoutId(linkStruct);
313             }
314             throw new Exception("The passed argument is not a link");
315         }
316     }
```

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

```
1   using System.Collections.Generic;
2   using Platform.Communication.Protocol.Lino;
3
4   namespace Platform.Data.Doublets.Lino;
5
6   public class LinoExporter<TLinkAddress>
7   {
8       private readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
9       private readonly ILinoStorage<TLinkAddress> _linoDocumentsStorage;
10
11      public LinoExporter(ILinoStorage<TLinkAddress> linoDocumentsStorage)
12      {
13          _linoDocumentsStorage = linoDocumentsStorage;
14      }
15
16      public string GetAllLinks(string? documentName)
17      {
18          var allLinks = _linoDocumentsStorage.GetLinks(documentName);
19          return allLinks.Format();
20      }
21  }
```

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

```
1   using System;
2   using System.IO;
3   using Platform.Data.Doublets.Memory.United.Generic;
4   using Platform.Data.Doublets.Sequences.Converters;
5   using Platform.IO;
6   using Platform.Memory;
7
8   namespace Platform.Data.Doublets.Lino;
9
10  public class LinoExporterCli<TLinkAddress> where TLinkAddress : struct
11  {
12      public void Run(params string[] args)
13      {
14          var argumentsIndex = 0;
15          var storageFilePath = ConsoleHelpers.GetOrReadArgument(argumentsIndex++, "A path to a
            ↪  links storage", args);
16          var notationFilePath = ConsoleHelpers.GetOrReadArgument(argumentsIndex++, "A path to a
            ↪  notation file", 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 exporter = new LinoExporter<TLinkAddress>(linoStorage);
30          var allLinksNotation = exporter.GetAllLinks(documentName);
31          File.WriteAllText(notationFilePath, allLinksNotation);
32      }
33  }
```

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, string? documentName)
    {
        var linoLinks = _parser.Parse(content);
        _linoDocumentsStorage.CreateLinks(linoLinks, documentName);
    }

}
```

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);
        var storageFilePath = ConsoleHelpers.GetOrReadArgument(argumentsIndex++, "A path to a links storage", 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 importer = new LinoImporter<TLinkAddress>(linoStorage);
        var notation = File.ReadAllText(notationFilePath);
        importer.Import(notation, documentName);
    }
}
```

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

```
using System;
using System.IO;
using Platform.IO;
using Xunit;
using TLinkAddress = System.UInt64;

namespace Platform.Data.Doublets.Lino.Tests;

public class ImporterAndExporterCliTests
{
    [Theory]
    [InlineData("(1: 1 1)")]
    [InlineData("(1: 1 1)\n(2: 2 2)")]
    [InlineData("(1: 2 2)")]
    [InlineData("(1: 2 2)\n(2: 1 1)")]
```

```
    public void DefaultLinoStorageTest(string notation)
    {
        var notationFilePath = TemporaryFiles.UseNew();
        var linksStorageFilePath = TemporaryFiles.UseNew();
        var exportedNotationFilePath = TemporaryFiles.UseNew();
        File.WriteAllText(notationFilePath, notation);
        new LinoImporterCli<TLinkAddress>().Run(notationFilePath, linksStorageFilePath, "");
        new LinoExporterCli<TLinkAddress>().Run(linksStorageFilePath, exportedNotationFilePath,
        ↪  "");
        Assert.Equal(notation, File.ReadAllText(exportedNotationFilePath));
    }

    [Theory]
    [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))")]
    public void LinoDocumentsStorageTest(string notation)
    {
        var notationFilePath = TemporaryFiles.UseNew();
        var linksStorageFilePath = TemporaryFiles.UseNew();
        var exportedNotationFilePath = TemporaryFiles.UseNew();
        File.WriteAllText(notationFilePath, notation);
        new LinoImporterCli<TLinkAddress>().Run(notationFilePath, linksStorageFilePath,
        ↪  notationFilePath);
        new LinoExporterCli<TLinkAddress>().Run(linksStorageFilePath, exportedNotationFilePath,
        ↪  notationFilePath);
        Assert.Equal(notation, File.ReadAllText(exportedNotationFilePath));
    }
}
```

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 System;
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);
        var documentName = Random.RandomHelpers.Default.Next(Int32.MaxValue).ToString();
        importer.Import(notation, documentName);
        var exporter = new LinoExporter<TLinkAddress>(linoStorage);
        var exportedLinks = exporter.GetAllLinks(documentName);
        Assert.Equal(notation, exportedLinks);
    }
```

```
37
38      [InlineData("(1: 1 1)")]
39      [InlineData("(1: 1 1)\n(2: 2 2)")]
40      [InlineData("(2: 2 2)")]
41      [InlineData("(1: 2 2)")]
42      [InlineData("(1: 2 2)\n(2: 1 1)")]
43      [InlineData("(1: 2 (3: 3 3))")]
44      [InlineData("(1: 2 (3: 3 3))\n(2: 1 1)")]
45      [InlineData("(son: lovesMama)")]
46      [InlineData("(papa: (lovesMama: loves mama))")]
47      [InlineData(@"(papa (lovesMama: loves mama))
48  (son lovesMama)
49  (daughter lovesMama)
50  (all (love mama))")]
51      [Theory]
52      public void LinoDocumentStorageTest(string notation)
53      {
54          var storage = CreateLinks();
55          var linoStorage = new LinoDocumentsStorage<TLinkAddress>(storage, new
            ↪  BalancedVariantConverter<ulong>(storage));
56          var importer = new LinoImporter<TLinkAddress>(linoStorage);
57          var documentName = Random.RandomHelpers.Default.Next(Int32.MaxValue).ToString();
58          importer.Import(notation, documentName);
59          var exporter = new LinoExporter<TLinkAddress>(linoStorage);
60          var exportedLinks = exporter.GetAllLinks(documentName);
61          Assert.Equal(notation, exportedLinks);
62      }
63
64  }
```

Index