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> where TLinkAddress :
↪  struct
{
    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 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;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.HeightProviders;
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;

```

```
17  namespace Platform.Data.Doublets.Lino;
18
19  public class LinoDocumentsStorage<TLinkAddress> : ILinoStorage<TLinkAddress> where TLinkAddress
    ↪  : struct
20  {
21          private readonly TLinkAddress _any;
22          private static readonly TLinkAddress Zero = default;
23          private static readonly TLinkAddress One = Arithmetic.Increment(Zero);
24
25          // public readonly IConverter<IList<TLinkAddress>?, TLinkAddress>
            ↪  ListToSequenceConverter;
26          private readonly TLinkAddress MeaningRoot;
27          private readonly EqualityComparer<TLinkAddress> _equalityComparer =
            ↪  EqualityComparer<TLinkAddress>.Default;
28          // Converters that are able to convert link's address (UInt64 value) to a raw number
            ↪  represented with another UInt64 value and back
29          public readonly RawNumberToAddressConverter<TLinkAddress> NumberToAddressConverter =
            ↪  new();
30          public readonly AddressToRawNumberConverter<TLinkAddress> AddressToNumberConverter =
            ↪  new();
31          // Converters between BigInteger and raw number sequence
32          public readonly BigIntegerToRawNumberSequenceConverter<TLinkAddress>
            ↪  BigIntegerToRawNumberSequenceConverter;
33          public readonly RawNumberSequenceToBigIntegerConverter<TLinkAddress>
            ↪  RawNumberSequenceToBigIntegerConverter;
34          // Converters between decimal and rational number sequence
35          public readonly DecimalToRationalConverter<TLinkAddress> DecimalToRationalConverter;
36          public readonly RationalToDecimalConverter<TLinkAddress> RationalToDecimalConverter;
37          public readonly DefaultSequenceRightHeightProvider<TLinkAddress>
            ↪  DefaultSequenceRightHeightProvider;
38          public readonly DefaultSequenceAppender<TLinkAddress> DefaultSequenceAppender;
39          public ILinks<TLinkAddress> Storage { get; }
40
41          public IConverter<string, TLinkAddress> StringToUnicodeSequenceConverter { get; }
42          public IConverter<TLinkAddress, string> UnicodeSequenceToStringConverter { get; }
43          public TLinkAddress DocumentMarker { get; }
44
45          public TLinkAddress ReferenceMarker { get; }
46          public TLinkAddress LinkMarker { get; }
47          public TLinkAddress LinkWithReferenceMarker { get; }
48          private TLinkAddress _markerIndex { get; set; }
49
50          private IConverter<IList<TLinkAddress>?, TLinkAddress> _listToSequenceConverter;
51
52          public LinoDocumentsStorage(ILinks<TLinkAddress> storage,
            ↪  IConverter<IList<TLinkAddress>?, TLinkAddress> listToSequenceConverter)
53          {
54              Storage = storage;
55              // ListToSequenceConverter = listToSequenceConverter;
56              // Initializes constants
57              _any = storage.Constants.Any;
58              var markerIndex = One;
59              MeaningRoot = storage.GetOrCreate(markerIndex, markerIndex);
60              var unicodeSymbolMarker = storage.GetOrCreate(MeaningRoot, Arithmetic.Increment(ref
                ↪  markerIndex));
61              var unicodeSequenceMarker = storage.GetOrCreate(MeaningRoot,
                ↪  Arithmetic.Increment(ref markerIndex));
62              DocumentMarker = storage.GetOrCreate(MeaningRoot, Arithmetic.Increment(ref
                ↪  markerIndex));
63              ReferenceMarker = storage.GetOrCreate(MeaningRoot, Arithmetic.Increment(ref
                ↪  markerIndex));
64              LinkMarker = storage.GetOrCreate(MeaningRoot, Arithmetic.Increment(ref markerIndex));
65              BalancedVariantConverter<TLinkAddress> balancedVariantConverter = new(storage);
66              TargetMatcher<TLinkAddress> unicodeSymbolCriterionMatcher = new(storage,
                ↪  unicodeSymbolMarker);
67              TargetMatcher<TLinkAddress> unicodeSequenceCriterionMatcher = new(storage,
                ↪  unicodeSequenceMarker);
68              CharToUnicodeSymbolConverter<TLinkAddress> charToUnicodeSymbolConverter =
69                  new(storage, AddressToNumberConverter, unicodeSymbolMarker);
70              UnicodeSymbolToCharConverter<TLinkAddress> unicodeSymbolToCharConverter =
71                  new(storage, NumberToAddressConverter, unicodeSymbolCriterionMatcher);
72              StringToUnicodeSequenceConverter = new CachingConverterDecorator<string,
                ↪  TLinkAddress>(
73                  new StringToUnicodeSequenceConverter<TLinkAddress>(storage,
                    ↪  charToUnicodeSymbolConverter,
74                      balancedVariantConverter, unicodeSequenceMarker));
75              RightSequenceWalker<TLinkAddress> sequenceWalker =
76                  new(storage, new DefaultStack<TLinkAddress>(),
                    ↪  unicodeSymbolCriterionMatcher.IsMatched);
```

```
77              UnicodeSequenceToStringConverter = new CachingConverterDecorator<TLinkAddress,
                ↪ string>(
78                  new UnicodeSequenceToStringConverter<TLinkAddress>(storage,
                    ↪ unicodeSequenceCriterionMatcher, sequenceWalker,
79                      unicodeSymbolToCharConverter));
80              DecimalToRationalConverter = new(storage, BigIntegerToRawNumberSequenceConverter);
81              RationalToDecimalConverter = new(storage, RawNumberSequenceToBigIntegerConverter);
82              _listToSequenceConverter = listToSequenceConverter;
83          }
84
85          public TLinkAddress GetOrCreateReferenceLink(string content) =>
            ↪ Storage.GetOrCreate(ReferenceMarker,
            ↪ StringToUnicodeSequenceConverter.Convert(content));
86
87          public string ReadReference(TLinkAddress reference) =>
            ↪ ReadReference(Storage.GetLink(reference));
88
89          public string ReadReference(IList<TLinkAddress> reference)
90          {
91              var referenceLink = new Link<TLinkAddress>(reference);
92              if (!_equalityComparer.Equals(ReferenceMarker, referenceLink.Source))
93              {
94                  throw new ArgumentException("The passed link is not a reference");
95              }
96              return UnicodeSequenceToStringConverter.Convert(referenceLink.Target);
97          }
98
99          public bool IsReference(TLinkAddress reference) =>
            ↪ IsReference(Storage.GetLink(reference));
100
101         public bool IsReference(IList<TLinkAddress> reference) =>
            ↪ _equalityComparer.Equals(ReferenceMarker, Storage.GetSource(reference));
102
103         public void CreateLinks(IList<LinoLink> links)
104         {
105             var sequenceList = new List<TLinkAddress>();
106             for (int i = 0; i < links.Count; i++)
107             {
108                 sequenceList.Add(CreateLink(links[i]));
109             }
110             Storage.GetOrCreate(DocumentMarker, _listToSequenceConverter.Convert(sequenceList));
111         }
112
113         public TLinkAddress CreateLink(LinoLink link)
114         {
115             if (link.Id != null)
116             {
117                 return CreateLinkWithReference(link);
118             }
119             var valuesSequence = CreateValuesSequence(link);
120             return Storage.GetOrCreate(LinkMarker, valuesSequence);
121         }
122
123         private TLinkAddress CreateLinkWithReference(LinoLink link)
124         {
125             TLinkAddress currentReference = GetOrCreateReferenceLink(link.Id);
126             if (link.Values == null)
127             {
128                 return currentReference;
129             }
130             var valuesSequence = CreateValuesSequence(link);
131             var idWithValues = Storage.GetOrCreate(currentReference, valuesSequence);
132             return Storage.GetOrCreate(LinkWithReferenceMarker, idWithValues);
133         }
134
135         public TLinkAddress CreateValuesSequence(LinoLink parent)
136         {
137             var values = new List<TLinkAddress>(parent.Values.Count);
138             for (int i = 0; i < parent.Values.Count; i++)
139             {
140                 var currentValue = parent.Values[i];
141                 if (currentValue.Values != null)
142                 {
143                     var valueLink = CreateLink(currentValue);
144                     values.Add(valueLink);
145                     continue;
146                 }
147                 var currentValueReference = GetOrCreateReferenceLink(currentValue.Id);
```

```
148                 values.Add(currentValueReference);
149             }
150             return _listToSequenceConverter.Convert(values);
151         }
152 
153         public IList<LinoLink> GetLinks()
154         {
155             var resultLinks = new List<LinoLink>();
156             var any = Storage.Constants.Any;
157             bool IsElement(TLinkAddress linkIndex)
158             {
159                 var source = Storage.GetSource(linkIndex);
160                 return _equalityComparer.Equals(LinkMarker, source) |
                    ↪  _equalityComparer.Equals(LinkWithReferenceMarker, source) ||
                    ↪  Storage.IsPartialPoint(linkIndex);
161             }
162             var rightSequenceWalker = new RightSequenceWalker<TLinkAddress>(Storage, new
                ↪  DefaultStack<TLinkAddress>(), IsElement);
163             TLinkAddress linksSequence = default;
164             Storage.Each(DocumentMarker, any, link =>
165             {
166                 linksSequence = Storage.GetTarget((IList<TLinkAddress>)link);
167                 return Storage.Constants.Continue;
168             });
169             if (_equalityComparer.Equals(default, linksSequence))
170             {
171                 throw new Exception("No one link in storage.");
172             }
173             var sequence = rightSequenceWalker.Walk(linksSequence);
174             foreach (var documentLink in sequence)
175             {
176                 resultLinks.Add(GetLink(documentLink));
177             }
178             return resultLinks;
179         }
180 
181         public LinoLink GetLink(TLinkAddress link) => GetLink(Storage.GetLink(link));
182 
183         public LinoLink GetLink(IList<TLinkAddress> link)
184         {
185             string id = default;
186             var values = new List<LinoLink>();
187             var linkStruct = new Link<TLinkAddress>(link);
188             var isLink = _equalityComparer.Equals(LinkMarker, linkStruct.Source) ||
                ↪  _equalityComparer.Equals(LinkWithReferenceMarker, linkStruct.Source);
189             if (!isLink)
190             {
191                 throw new Exception("The source of the passed link is not the link marker.");
192             }
193             bool IsElement(TLinkAddress linkIndex)
194             {
195                 var source = Storage.GetSource(linkIndex);
196                 return _equalityComparer.Equals(LinkMarker, source) ||
                    ↪  _equalityComparer.Equals(LinkWithReferenceMarker, source) ||
                    ↪  _equalityComparer.Equals(ReferenceMarker, source) ||
                    ↪  Storage.IsPartialPoint(linkIndex);
197             }
198             var rightSequenceWalker = new RightSequenceWalker<TLinkAddress>(Storage, new
                ↪  DefaultStack<TLinkAddress>(), IsElement);
199             foreach (var currentLink in rightSequenceWalker.Walk(linkStruct.Target))
200             {
201                 var currentLinkStruct = new Link<TLinkAddress>(Storage.GetLink(currentLink));
202                 if (_equalityComparer.Equals(LinkMarker, currentLinkStruct.Source) ||
                    ↪  _equalityComparer.Equals(LinkWithReferenceMarker, currentLinkStruct.Source))
203                 {
204                     var value = GetLink(currentLinkStruct);
205                     values.Add(value);
206                 }
207                 else if (_equalityComparer.Equals(ReferenceMarker, currentLinkStruct.Source))
208                 {
209                     if (default == id)
210                     {
211                         id = ReadReference(currentLinkStruct);
212                     }
213                     else
214                     {
215                         var currentLinoLink = new LinoLink(ReadReference(currentLinkStruct));
216                         values.Add(currentLinoLink);
```

```
                    }
                }

            }
            return new LinoLink(id, values);
        }
}
```

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> where TLinkAddress : struct
{
    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();
    }

    // public void GetAllLinks(Stream outputStream)
    // {
    //     var allLinks = _linoDocumentsStorage.Storage.All();
    //     var linksAsStrings = new Dictionary<TLinkAddress, string>(allLinks.Count);
    //     for (int i = 0; i < allLinks.Count; i++)
    //     {
    //         var currentLink = allLinks[i];
    //         linksAsStrings.Add(_linoDocumentsStorage.Storage.GetIndex(currentLink), ReadLinkAsString(currentLink));
    //     }
    // }
    //
    // private LinoLink ReadLinkAsString(IList<TLinkAddress> link)
    // {
    //     string source = "";
    //     string target = "";
    //     var linkStruct = new Link<TLinkAddress>(link);
    //     if (_linoDocumentsStorage.IsReference(linkStruct.Source))
    //     {
    //         source = _linoDocumentsStorage.ReadReference(linkStruct.Source);
    //     }
    //     if (_linoDocumentsStorage.IsReference(linkStruct.Target))
    //     {
    //         target = _linoDocumentsStorage.ReadReference(linkStruct.Target);
    //     }
    //     return new LinoLink(source, target);
    // }
}
```

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

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

namespace Platform.Data.Doublets.Lino;

public class LinoImporter<TLinkAddress> where TLinkAddress : struct
{
    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);
    }
```

```
19
20      // public void Import(string content)
21      // {
22      //     var linoLinks = _parser.Parse(content);
23      //     for (int i = 0; i < linoLinks.Count; i++)
24      //     {
25      //         var linoLink = linoLinks[i];
26      //         Read(linoLink);
27      //     }
28      // }
29      //
30      // public TLinkAddress Read(Link parent)
31      // {
32      //     var left = parent.Values[0];
33      //     var right = parent.Values[1];
34      //     var source = left.Values != null ? Read(left) :
        ↪  _linoDocumentsStorage.GetOrCreateReferenceLink(left.Id);
35      //     var target = right.Values != null ? Read(right) :
        ↪  _linoDocumentsStorage.GetOrCreateReferenceLink(right.Id);
36      //     return _linoDocumentsStorage.Storage.GetOrCreate(source, target);
37      // }
38  }
```

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

```
1   using Platform.Data.Doublets.Memory.United.Generic;
2   using Platform.IO;
3   using Platform.Memory;
4
5   namespace Platform.Data.Doublets.Lino;
6
7   public class LinoImporterCli<TLinkAddress>
8   {
9       public void Run(params string[] args)
10      {
11          var argumentsIndex = 0;
12          // var notationFilePath = ConsoleHelpers.GetOrReadArgument(argumentsIndex++, "Path to a
            ↪  notation file", args);
13          // var storageFilePath = ConsoleHelpers.GetOrReadArgument(argumentsIndex, "Path to a
            ↪  links storage", args);
14          // var linksMemory = new FileMappedResizableDirectMemory(storageFilePath);
15          // var links = new UnitedMemoryLinks<TLinkAddress>(linksMemory).DecorateWithAutomaticUni
            ↪  quenessAndUsagesResolution();
16          // var linoStorage = new DefaultLinoStorage<TLinkAddress>();
17          // var importer = new LinoImporter<TLinkAddress>()
18      }
19  }
```

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

```
1   using Platform.Converters;
2   using Platform.Data.Doublets.CriterionMatchers;
3   using Platform.Data.Doublets.Memory;
4   using Platform.Data.Doublets.Memory.United.Generic;
5   using Platform.Data.Doublets.Sequences.Converters;
6   using Platform.Data.Doublets.Unicode;
7   using Platform.Data.Numbers.Raw;
8   using Platform.Memory;
9   using Platform.Numbers;
10  using Xunit;
11  using TLinkAddress = System.UInt64;
12
13  namespace Platform.Data.Doublets.Lino.Tests;
14
15  public class ImporterAndExporterTests
16  {
17      public static ILinks<TLinkAddress> CreateLinks() => CreateLinks(new IO.TemporaryFile());
18
19      public static ILinks<TLinkAddress> CreateLinks(string dbFilename)
20      {
21          var linksConstants = new LinksConstants<TLinkAddress>(enableExternalReferencesSupport:
            ↪  true);
22          return new UnitedMemoryLinks<TLinkAddress>(new HeapResizableDirectMemory(),
            ↪  UnitedMemoryLinks<TLinkAddress>.DefaultLinksSizeStep, linksConstants,
            ↪  IndexTreeType.Default);
23      }
24
25      // [InlineData("(1: 1 1)")]
26      // [InlineData("(1: 1 1)\n(2: 2 2)")]
27      // [InlineData("(1: 2 2)\n(2: 1 1)")]
28      // [Theory]
```

```
29      // public void Test1(string notation)
30      // {
31      //     var storage = CreateLinks();
32      //     TLinkAddress Zero = default;
33      //     TLinkAddress One = Arithmetic.Increment(Zero);
34      //     var markerIndex = One;
35      //     var meaningRoot = storage.GetOrCreate(markerIndex, markerIndex);
36      //     var unicodeSymbolMarker = storage.GetOrCreate(meaningRoot, Arithmetic.Increment(ref
        ↪  markerIndex));
37      //     var unicodeSequenceMarker = storage.GetOrCreate(meaningRoot, Arithmetic.Increment(ref
        ↪  markerIndex));
38      //     var referenceMarker = storage.GetOrCreate(meaningRoot, Arithmetic.Increment(ref
        ↪  markerIndex));
39      //     TargetMatcher<TLinkAddress> unicodeSymbolCriterionMatcher = new(storage,
        ↪  unicodeSymbolMarker);
40      //     TargetMatcher<TLinkAddress> unicodeSequenceCriterionMatcher = new(storage,
        ↪  unicodeSequenceMarker);
41      //     AddressToRawNumberConverter<TLinkAddress> addressToNumberConverter = new();
42      //     RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
43      //     CharToUnicodeSymbolConverter<TLinkAddress> charToUnicodeSymbolConverter =
44      //         new(storage, addressToNumberConverter, unicodeSymbolMarker);
45      //     UnicodeSymbolToCharConverter<TLinkAddress> unicodeSymbolToCharConverter =
46      //         new(storage, numberToAddressConverter, unicodeSymbolCriterionMatcher);
47      //     var balancedVariantConverter = new BalancedVariantConverter<TLinkAddress>(storage);
48      //     var stringToUnicodeSequenceConverter = new CachingConverterDecorator<string,
        ↪  TLinkAddress>(new StringToUnicodeSequenceConverter<TLinkAddress>(storage,
        ↪  charToUnicodeSymbolConverter, balancedVariantConverter, unicodeSequenceMarker));
49      //     ILinoDocumentsStorage<TLinkAddress> linoDocumentsStorage = new
        ↪  LinoDocumentsStorage<TLinkAddress>(storage);
50      //     LinoImporter<TLinkAddress> linoImporter = new(linoDocumentsStorage);
51      //     linoImporter.Import(notation);
52      //     var linoExporter = new LinoExporter<TLinkAddress>(linoDocumentsStorage);
53      // }
54
55      [InlineData("(1: 1 1)")]
56      [InlineData("(1: 1 1)\n(2: 2 2)")]
57      [InlineData("(1: 2 2)")]
58      [InlineData("(1: 2 2)\n(2: 1 1)")]
59      [Theory]
60      public void LinoStorageTest(string notation)
61      {
62          var storage = CreateLinks();
63          var linoStorage = new DefaultLinoStorage<TLinkAddress>(storage);
64          var importer = new LinoImporter<TLinkAddress>(linoStorage);
65          importer.Import(notation);
66          var exporter = new LinoExporter<TLinkAddress>(linoStorage);
67          var exportedLinks = exporter.GetAllLinks();
68          Assert.Equal(notation, exportedLinks);
69      }
70
71      [InlineData("(1: 1 1)")]
72      [InlineData("(1: 1 1)\n(2: 2 2)")]
73      [InlineData("(2: 2 2)")]
74      [InlineData("(1: 2 2)")]
75      [InlineData("(1: 2 2)\n(2: 1 1)")]
76      [InlineData("(1: 2 (3: 3 3))")]
77      [InlineData("(1: 2 (3: 3 3))\n(2: 1 1)")]
78      [InlineData("(son: lovesMama)")]
79      [InlineData("(papa: (lovesMama: loves mama))")]
80      [InlineData("(papa: (lovesMama: loves mama) son lovesMama daughter lovesMama all (love:
    ↪  mama))")]
81      [Theory]
82      public void LinoDocumentStorageTest(string notation)
83      {
84          var storage = CreateLinks();
85          var linoStorage = new LinoDocumentsStorage<TLinkAddress>(storage, new
            ↪  BalancedVariantConverter<ulong>(storage));
86          var importer = new LinoImporter<TLinkAddress>(linoStorage);
87          importer.Import(notation);
88          var exporter = new LinoExporter<TLinkAddress>(linoStorage);
89          var exportedLinks = exporter.GetAllLinks();
90          Assert.Equal(notation, exportedLinks);
91      }
92
93      // public void CreateLinksTest()
94      // {
95      //     var storage = CreateLinks();
```

```
96      //      var linoStorage = new DefaultLinoStorage<TLinkAddress>(storage);
97      //      linoStorage.CreateLinks();
98      // }
99      //
100     // public void GetLinksTest()
101     // {
102     //      var storage = CreateLinks();
103     //      var linoStorage = new DefaultLinoStorage<TLinkAddress>(storage);
104     //      linoStorage.GetLinks();
105     // }
106 }
```

Index