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;

```

```
namespace Platform.Data.Doublets.Lino;

public class LinoDocumentsStorage<TLinkAddress> : ILinoStorage<TLinkAddress> where TLinkAddress
↪  : struct
{
        private readonly TLinkAddress _any;
        private static readonly TLinkAddress Zero = default;
        private static readonly TLinkAddress One = Arithmetic.Increment(Zero);

        // public readonly IConverter<IList<TLinkAddress>?, TLinkAddress>
        ↪  ListToSequenceConverter;
        private readonly TLinkAddress MeaningRoot;
        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
        public readonly RawNumberToAddressConverter<TLinkAddress> NumberToAddressConverter =
        ↪  new();
        public readonly AddressToRawNumberConverter<TLinkAddress> AddressToNumberConverter =
        ↪  new();
        // Converters between BigInteger and raw number sequence
        public readonly BigIntegerToRawNumberSequenceConverter<TLinkAddress>
        ↪  BigIntegerToRawNumberSequenceConverter;
        public readonly RawNumberSequenceToBigIntegerConverter<TLinkAddress>
        ↪  RawNumberSequenceToBigIntegerConverter;
        // Converters between decimal and rational number sequence
        public readonly DecimalToRationalConverter<TLinkAddress> DecimalToRationalConverter;
        public readonly RationalToDecimalConverter<TLinkAddress> RationalToDecimalConverter;
        public readonly DefaultSequenceRightHeightProvider<TLinkAddress>
        ↪  DefaultSequenceRightHeightProvider;
        public readonly DefaultSequenceAppender<TLinkAddress> DefaultSequenceAppender;
        public ILinks<TLinkAddress> Storage { get; }

        public IConverter<string, TLinkAddress> StringToUnicodeSequenceConverter { get; }
        public IConverter<TLinkAddress, string> UnicodeSequenceToStringConverter { get; }
        public TLinkAddress DocumentMarker { get; }

        public TLinkAddress ReferenceMarker { get; }
        public TLinkAddress LinkMarker { get; }
        private TLinkAddress _markerIndex { get; set; }

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

        public LinoDocumentsStorage(ILinks<TLinkAddress> storage,
        ↪  IConverter<IList<TLinkAddress>?, TLinkAddress> listToSequenceConverter)
        {
            Storage = storage;
            // ListToSequenceConverter = listToSequenceConverter;
            // Initializes constants
            _any = storage.Constants.Any;
            var markerIndex = One;
            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));
            LinkMarker = 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);
```

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

```
            {
                return _equalityComparer.Equals(LinkMarker, Storage.GetSource(linkIndex)) ||
                ↪  Storage.IsPartialPoint(linkIndex);
            }
            var rightSequenceWalker = new RightSequenceWalker<TLinkAddress>(Storage, new
            ↪  DefaultStack<TLinkAddress>(), IsElement);
            TLinkAddress linksSequence = default;
            Storage.Each(DocumentMarker, any, link =>
            {
                linksSequence = Storage.GetTarget((IList<TLinkAddress>)link);
                return Storage.Constants.Continue;
            });
            if (_equalityComparer.Equals(default, linksSequence))
            {
                throw new Exception("No one link in storage.");
            }
            var sequence = rightSequenceWalker.Walk(linksSequence);
            foreach (var documentLink in sequence)
            {
                resultLinks.Add(GetLink(documentLink));
            }
            return resultLinks;
        }

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

        public LinoLink GetLink(IList<TLinkAddress> link)
        {
            string id = default;
            var values = new List<LinoLink>();
            var linkStruct = new Link<TLinkAddress>(link);
            if (!_equalityComparer.Equals(LinkMarker, linkStruct.Source))
            {
                throw new Exception("The source of the passed link is not the link marker.");
            }
            bool IsElement(TLinkAddress linkIndex)
            {
                var source = Storage.GetSource(linkIndex);
                return _equalityComparer.Equals(LinkMarker, source) ||
                ↪  _equalityComparer.Equals(ReferenceMarker, source) ||
                ↪  Storage.IsPartialPoint(linkIndex);
            }
            var rightSequenceWalker = new RightSequenceWalker<TLinkAddress>(Storage, new
            ↪  DefaultStack<TLinkAddress>(), IsElement);
            foreach (var currentLink in rightSequenceWalker.Walk(linkStruct.Target))
            {
                var currentLinkStruct = new Link<TLinkAddress>(Storage.GetLink(currentLink));
                if (_equalityComparer.Equals(LinkMarker, currentLinkStruct.Source))
                {
                    var value = GetLink(currentLinkStruct);
                    values.Add(value);
                    continue;
                }
                if (_equalityComparer.Equals(ReferenceMarker, currentLinkStruct.Source))
                {
                    if (default == id)
                    {
                        id = ReadReference(currentLinkStruct);
                    }
                    else
                    {
                        var currentLinoLink = new LinoLink(ReadReference(currentLinkStruct));
                        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);
    }

    // public void Import(string content)
    // {
    //     var linoLinks = _parser.Parse(content);
    //     for (int i = 0; i < linoLinks.Count; i++)
    //     {
    //         var linoLink = linoLinks[i];
    //         Read(linoLink);
    //     }
    // }
    //
    // public TLinkAddress Read(Link parent)
    // {
    //     var left = parent.Values[0];
    //     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.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
            ↪  FileMappedResizableDirectMemory(dbFilename),
            ↪  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)\n(2: 1 1)")]
```

```
58      [Theory]
59      public void LinoStorageTest(string notation)
60      {
61          var storage = CreateLinks();
62          var linoStorage = new DefaultLinoStorage<TLinkAddress>(storage);
63          var importer = new LinoImporter<TLinkAddress>(linoStorage);
64          importer.Import(notation);
65          var exporter = new LinoExporter<TLinkAddress>(linoStorage);
66          var exportedLinks = exporter.GetAllLinks();
67          Assert.Equal(notation, exportedLinks);
68      }
69
70      [InlineData("(1: 1 1)")]
71      [InlineData("(1: 1 1)\n(2: 2 2)")]
72      [InlineData("(2: 2 2)")]
73      [InlineData("(1: 2 2)")]
74      [InlineData("(1: 2 2)\n(2: 1 1)")]
75      [InlineData("(1: 2 (3: 3 3))")]
76      [InlineData("(1: 2 (3: 3 3))\n(2: 1 1)")]
77      [Theory]
78      public void LinoDocumentStorageTest(string notation)
79      {
80          var storage = CreateLinks();
81          var linoStorage = new LinoDocumentsStorage<TLinkAddress>(storage, new
            ↪  BalancedVariantConverter<ulong>(storage));
82          var importer = new LinoImporter<TLinkAddress>(linoStorage);
83          importer.Import(notation);
84          var anotherLinoStorage = new DefaultLinoStorage<TLinkAddress>(storage);
85          // var exporter = new LinoExporter<TLinkAddress>(anotherLinoStorage);
86          var exporter = new LinoExporter<TLinkAddress>(linoStorage);
87          var exportedLinks = exporter.GetAllLinks();
88          Assert.Equal(notation, exportedLinks);
89      }
90
91      // public void CreateLinksTest()
92      // {
93      //     var storage = CreateLinks();
94      //     var linoStorage = new DefaultLinoStorage<TLinkAddress>(storage);
95      //     linoStorage.CreateLinks();
96      // }
97      //
98      // public void GetLinksTest()
99      // {
100     //     var storage = CreateLinks();
101     //     var linoStorage = new DefaultLinoStorage<TLinkAddress>(storage);
102     //     linoStorage.GetLinks();
103     // }
104 }
```

Index