C# LinkedList Implementation

Gabriel Martin

October 2, 2023

1 Introduction

This document presents a C# implementation of a doubly-linked list, along with a test suite for the list operations. The implementation includes a generic linked list class, a custom node class, and a derived class specialized for storing and managing instances of the `Person` class.

2 C# Code

2.1 LinkedList Implementation

```
1
2  using System.Collections;
3  using System.Text;
4
5  namespace LinkedListDLL;
6
7  public class MyLinkedList<T> : IEnumerable<T>
8  {
9          private int m_Size;
10         private Node<T>? m_Head;
11         private Node<T>? m_Tail;
12
13         private class Node<TDataType>
14         {
15                 public TDataType Data;
16                 public Node<TDataType>? Prev;
17                 public Node<TDataType>? Next;
18
19                 public Node(TDataType data, Node<TDataType>? prev, Node<TDataType>? next)
20                 {
21                         Data = data;
22                         Prev = prev;
23                         Next = next;
24                 }
25                 public override string? ToString()
```

```
26              {
27                      return Data?.ToString();
28              }
29      }
30
31      public T this[int index]
32      {
33              get
34              {
35                      int i;
36                      Node<T>? trav;
37                      if (index < 0 || index > m_Size) throw new IndexOutOfRangeException();
38                      if (index < m_Size / 2)
39                      {
40                              for (i = 0, trav = m_Head; i != index; i++)
41                                      trav = trav.Next;
42                      }
43                      else
44                      {
45                              for (i = m_Size - 1, trav = m_Tail; i != index; i--)
46                                      trav = trav?.Prev;
47                      }
48                      return trav!.Data;
49              }
50              set
51              {
52                      int i;
53                      Node<T>? trav;
54                      if (index < 0 || index > m_Size) throw new IndexOutOfRangeException();
55                      if (index < m_Size / 2)
56                      {
57                              for (i = 0, trav = m_Head; i != index; i++)
58                                      trav = trav?.Next;
59                      }
60                      else
61                      {
62                              for (i = m_Size - 1, trav = m_Tail; i != index; i--)
63                                      trav = trav?.Prev;
64                      }
65                      trav!.Data = value;
66              }
67      }
68
69      // TimeComplexity: O(n)
70      public void Clear()
71      {
```

```
72                  var trav = m_Head;
73                  while (trav != null)
74                  {
75                          Node<T>? next = trav.Next;
76                          trav.Prev = trav.Next = null;
77                          trav = next;
78                  }
79                  m_Head = m_Tail = trav = null;
80                  m_Size = 0;
81          }
82
83          public int Size() { return m_Size; }
84          public bool IsEmpty() { return m_Size == 0; }
85
86          public void Add(T value) { AddLast(value); }
87
88          public void AddFirst(T value)
89          {
90                  if (IsEmpty())
91                  {
92                          m_Head = m_Tail = new Node<T>(value, null, null);
93                  }
94                  else
95                  {
96                          m_Head!.Prev = new Node<T>(value, null, m_Head);
97                  }
98                  m_Size++;
99          }
100
101         public void AddLast(T value)
102         {
103                 if (IsEmpty())
104                 {
105                         m_Head = m_Tail = new Node<T>(value, null, null);
106                 }
107                 else
108                 {
109                         m_Tail!.Next = new Node<T>(value, m_Tail, null);
110                         m_Tail = m_Tail.Next;
111                 }
112                 m_Size++;
113         }
114
115         public T PeekFirst()
116         {
117                 if (IsEmpty()) throw new Exception("Empty list");
```

```
118             return m_Head!.Data;
119         }
120
121         public T PeekLast()
122         {
123             if (IsEmpty()) throw new Exception("Empty list");
124             return m_Tail!.Data;
125         }
126
127         public T RemoveFirst()
128         {
129             if (IsEmpty()) throw new Exception("Empty list");
130             var data = m_Head!.Data;
131             m_Head = m_Head.Next;
132             m_Size--;
133
134             if (IsEmpty()) m_Tail = null;
135             else m_Head!.Prev = null;
136             return data;
137         }
138
139         public T RemoveLast()
140         {
141             if (IsEmpty()) throw new Exception("Empty list");
142             var data = m_Tail!.Data;
143             m_Tail = m_Tail.Prev;
144             m_Size--;
145
146             if (IsEmpty()) m_Head = null;
147             else m_Tail!.Next = null;
148             return data;
149         }
150
151         private T Remove(Node<T>? node)
152         {
153             if (node?.Prev == null) return RemoveFirst();
154             if (node.Next == null) return RemoveLast();
155
156             node.Next.Prev = node.Prev;
157             node.Prev.Next = node.Next;
158
159             var data = node.Data;
160             node = node.Prev = node.Next = null;
161             m_Size--;
162             return data;
163         }
```

```
164
165         public T RemoveIndex(int index)
166         {
167                 if (index < 0 || index >= m_Size) throw new IndexOutOfRangeException();
168
169                 int i;
170                 Node<T>? trav;
171
172                 if (index < m_Size / 2)
173                 {
174                         for (i = 0, trav = m_Head; i != index; i++)
175                                 trav = trav?.Next;
176                 }
177                 else
178                 {
179                         for (i = m_Size - 1, trav = m_Tail; i != index; i--)
180                                 trav = trav?.Prev;
181                 }
182                 return Remove(trav);
183         }
184
185         public bool Remove(T value)
186         {
187                 if (value == null)
188                 {
189                         for (var trav = m_Head; trav != null; trav = trav.Next)
190                         {
191                                 if (trav.Data != null) continue;
192                                 Remove(trav);
193                                 return true;
194                         }
195                 }
196                 else
197                 {
198                         for (var trav = m_Head; trav != null; trav = trav.Next)
199                         {
200                                 if (!value.Equals(trav.Data)) continue;
201                                 Remove(trav);
202                                 return true;
203                         }
204                 }
205
206                 return false;
207         }
208
209         public int IndexOf(T value)
```

```
210         {
211                 var index = 0;
212                 if (value == null)
213                 {
214                         for (var trav = m_Head; trav != null; trav = trav.Next, index++)
215                                 if (trav.Data == null)
216                                         return index;
217                 }
218                 else
219                         for (var trav = m_Head; trav != null; trav = trav.Next, index++)
220                                 if (value.Equals(trav.Data))
221                                         return index;
222                 return -1;
223         }
224
225         public bool Contains(T value)
226         {
227                 return IndexOf(value) != -1;
228         }
229
230         public IEnumerator<T> GetEnumerator()
231         {
232                 var trav = m_Head;
233                 while (trav != null)
234                 {
235                         yield return trav.Data;
236                         trav = trav.Next;
237                 }
238         }
239
240         IEnumerator IEnumerable.GetEnumerator()
241         {
242                 return this.GetEnumerator();
243         }
244
245         public override string ToString()
246         {
247                 var output = new StringBuilder("[ ");
248                 var trav = m_Head;
249                 while (trav != null)
250                 {
251                         output.Append(trav.Data);
252                         output.Append(", ");
253                         trav = trav.Next;
254                 }
255
```

```
                try
                {
                        output.Remove(output.Length - 2, 2);
                }
                catch
                {
                        // ignored
                }

                output.Append(" ]");
                return output.ToString();
        }
}
```

2.2 Person Class

```
namespace LinkedListDLL;

public class Person
{
    public string Name { get; }
    public int Age { get; }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public override bool Equals(object? obj)
    {
        if (obj == null || GetType() != obj.GetType())
            return false;

        var other = (Person)obj;
        return Name == other.Name && Age == other.Age;
    }

    protected bool Equals(Person other)
    {
        return Name == other.Name && Age == other.Age;
    }

    public override int GetHashCode()
```

```
    {
        return HashCode.Combine(Name, Age);
    }

    public override string ToString()
    {
        return Name;
    }
}
```

2.3 PersonLinkedList Class

```
namespace LinkedListDLL;

public class PersonLinkedList : MyLinkedList<Person>
{
    public Person? Find(string name)
    {
        return this.FirstOrDefault(person => person.Name == name);
    }

    public string FindMin()
    {
        if (IsEmpty()) throw new Exception("List is empty");

        var minName = PeekFirst().Name;
        foreach (var person in this)
            if (string.Compare(person.Name, minName, StringComparison.OrdinalIgnoreCase) < 0)
                minName = person.Name;

        return minName;
    }

    public string FindMax()
    {
        if (IsEmpty()) throw new Exception("List is empty");

        var maxName = PeekFirst().Name;
        foreach (var person in this)
            if (string.Compare(person.Name, maxName, StringComparison.OrdinalIgnoreCase) > 0)
                maxName = person.Name;

        return maxName;
    }

    public Person FindYoungest()
```

```
    {
        if (IsEmpty()) throw new Exception("List is empty");

        var youngest = PeekFirst();
        foreach (var person in this)
            if (person.Age < youngest.Age)
                youngest = person;

        return youngest;
    }

    public Person FindOldest()
    {
        if (IsEmpty())
        {
            throw new Exception("List is empty");
        }

        var oldest = PeekFirst();
        foreach (var person in this)
            if (person.Age > oldest.Age)

                oldest = person;

        return oldest;
    }
}
```

2.4 Test Suite

```
using System.Diagnostics;
using LinkedListDLL;

namespace LinkedListTest;

[TestFixture]
public class PersonLinkedListTests
{
    private PersonLinkedList m_PersonList = null!;

    [SetUp]
    public void Setup()
    {
        m_PersonList = new PersonLinkedList();
        m_PersonList.AddLast(new Person("Alice", 30));
        m_PersonList.AddLast(new Person("Bob", 25));
```

```
17          m_PersonList.AddLast(new Person("Charlie", 35));
18          m_PersonList.AddLast(new Person("David", 28));
19      }
20
21      [Test]
22      public void TestBasicOperations()
23      {
24          Assert.Multiple(() =>
25          {
26              Assert.That(m_PersonList.Size(), Is.EqualTo(4));
27              Assert.That(m_PersonList.IsEmpty(), Is.False);
28          });
29      }
30
31      [Test]
32      public void TestFind()
33      {
34          var foundPerson = m_PersonList.Find("Bob");
35          Assert.That(foundPerson, Is.Not.Null);
36          Assert.That(foundPerson!.Name, Is.EqualTo("Bob"));
37      }
38
39      [Test]
40      public void TestFindMinAndMax()
41      {
42          Assert.Multiple(() =>
43          {
44              Assert.That(m_PersonList.FindMin(), Is.EqualTo("Alice"));
45              Assert.That(m_PersonList.FindMax(), Is.EqualTo("David"));
46          });
47      }
48
49      [Test]
50      public void TestFindYoungestAndOldest()
51      {
52          Assert.Multiple(() =>
53          {
54              Assert.That(m_PersonList.FindYoungest().Name, Is.EqualTo("Bob"));
55              Assert.That(m_PersonList.FindOldest().Name, Is.EqualTo("Charlie"));
56          });
57      }
58
59      [Test]
60      public void TestIndexingAndRemovalByIndex()
61      {
62          Assert.That(m_PersonList[2].Name, Is.EqualTo("Charlie"));
```

```
63          m_PersonList.RemoveIndex(2);
64          Assert.That(m_PersonList[2].Name, Is.EqualTo("David"));
65      }
66
67      [Test]
68      public void TestRemovalByValue()
69      {
70          var removed = m_PersonList.Remove(new Person("Charlie", 35));
71          Assert.Multiple(() =>
72          {
73              Assert.That(removed, Is.True);
74              Assert.That(m_PersonList.Contains(new Person("Charlie", 35)), Is.False);
75          });
76      }
77
78      [Test]
79      public void TestContains()
80      {
81          Assert.Multiple(() =>
82          {
83              Assert.That(m_PersonList.Contains(new Person("Alice", 30)), Is.True);
84              Assert.That(m_PersonList.Contains(new Person("Emily", 40)), Is.False);
85          });
86      }
87
88      [Test]
89      public void TestClear()
90      {
91          m_PersonList.Clear();
92          Assert.Multiple(() =>
93          {
94              Assert.That(m_PersonList.Size(), Is.EqualTo(0));
95              Assert.That(m_PersonList.IsEmpty());
96          });
97      }
98
99      [Test]
100     public void TestPeekFirstAndPeekLast()
101     {
102         Assert.Multiple(() =>
103         {
104             Assert.That(m_PersonList.PeekFirst().Name, Is.EqualTo("Alice"));
105             Assert.That(m_PersonList.PeekLast().Name, Is.EqualTo("David"));
106         });
107     }
108
```

```
109     [Test]
110     public void TestRemoveFirstAndRemoveLast()
111     {
112         Assert.Multiple(() =>
113         {
114             Assert.That(m_PersonList.RemoveFirst().Name, Is.EqualTo("Alice"));
115             Assert.That(m_PersonList.RemoveLast().Name, Is.EqualTo("David"));
116         });
117     }
118
119     [Test]
120     public void TestToString()
121     {
122         const string expected = "[ Alice, Bob, Charlie, David ]";
123         Assert.That(m_PersonList.ToString(), Is.EqualTo(expected));
124     }
125
126     [Test]
127     public void TestIndexingOutOfRange()
128     {
129         Assert.Throws<IndexOutOfRangeException>(() =>
130         {
131             var x = m_PersonList[10];
132         });
133     }
134 }
```

3 Explanation

Here is a brief explanation of the key components and operations in the provided C# code:

- **MyLinkedList¡T¿ Class:** This class represents a generic doubly-linked list. It includes methods for adding, removing, and manipulating elements in the list.
- **Node¡TDataType¿ Class:** This is a nested class within ‘MyLinkedList‘, representing a node in the linked list. It holds the data, as well as references to the previous and next nodes.
- **Person Class:** This class represents a simple ‘Person‘ with a name and age. It includes methods for equality comparison and hashing.
- **PersonLinkedList Class:** This class is derived from ‘MyLinkedList¡Person¿‘ and includes additional methods for finding the minimum, maximum, youngest, and oldest person in the list.
- **Test Suite:** The ‘PersonLinkedListTests‘ class contains unit tests for various operations on the ‘PersonLinkedList‘ class, ensuring that it functions correctly.