Deakin University

Data Structures and Algorithms

OnTrack Submission

Doubly Linked List

Submitted By:
Peter Stacey
pstacey
2020/08/20 17:29

Tutor:
Maksym Slavnenko

Outcome	**Weight**
Complexity	♦◇◇◇◇
Implement Solutions	♦♦♦◇◇
Document solutions	♦◇◇◇◇

The task involes understanding Doubly Linked List as a data structure and implementing methods that provide access to members of the list and to add and remove values from a linked list. This aligns with ULO2.

August 21, 2020

```
using System;
using System.Text;

namespace DoublyLinkedList
{
    public class DoublyLinkedList<T>
    {

        // Here is the the nested Node<K> class
        private class Node<K> : INode<K>
        {
            public K Value { get; set; }
            public Node<K> Next { get; set; }
            public Node<K> Previous { get; set; }

            public Node(K value, Node<K> previous, Node<K> next)
            {
                Value = value;
                Previous = previous;
                Next = next;
            }

            // This is a ToString() method for the Node<K>
            // It represents a node as a tuple {'the previous node's value'-(the node's value)-'the next node's value')}.
            // 'XXX' is used when the current node matches the First or the Last of the DoublyLinkedList<T>
            public override string ToString()
            {
                StringBuilder s = new StringBuilder();
                s.Append("{");
                s.Append(Previous.Previous == null ? "XXX" : Previous.Value.ToString());
                s.Append("-(");
                s.Append(Value);
                s.Append(")-");
                s.Append(Next.Next == null ? "XXX" : Next.Value.ToString());
                s.Append("}");
                return s.ToString();
            }

        }

        // Here is where the description of the methods and attributes of the DoublyLinkedList<T> class starts

        // An important aspect of the DoublyLinkedList<T> is the use of two auxiliary nodes: the Head and the Tail.
        // The both are introduced in order to significantly simplify the implementation of the class and make insertion functionality reduced just to a AddBetween(...)
        // These properties are private, thus are invisible to a user of the data structure, but are always maintained in it, even when the DoublyLinkedList<T> is formally empty.
```

```
46          // Remember about this crucial fact when you design and code other
            ↪  functions of the DoublyLinkedList<T> in this task.
47          private Node<T> Head { get; set; }
48          private Node<T> Tail { get; set; }
49          public int Count { get; private set; } = 0;
50
51          public DoublyLinkedList()
52          {
53              Head = new Node<T>(default(T), null, null);
54              Tail = new Node<T>(default(T), Head, null);
55              Head.Next = Tail;
56          }
57
58          public INode<T> First
59          {
60              get
61              {
62                  if (Count == 0) return null;
63                  else return Head.Next;
64              }
65          }
66
67          public INode<T> Last
68          {
69              get
70              {
71                  if (Count == 0) return null;
72                  else return Tail.Previous;
73              }
74          }
75
76          public INode<T> After(INode<T> node)
77          {
78              if (node == null) throw new NullReferenceException();
79              Node<T> node_current = node as Node<T>;
80              if (node_current.Previous == null || node_current.Next == null)
81                  throw new InvalidOperationException("The node referred as 'before'
                    ↪  is no longer in the list");
82              if (node_current.Next.Equals(Tail)) return null;
83              else return node_current.Next;
84          }
85
86          public INode<T> AddLast(T value)
87          {
88              return AddBetween(value, Tail.Previous, Tail);
89          }
90
91          // This is a private method that creates a new node and inserts it in
            ↪  between the two given nodes referred as the previous and the next.
92          // Use it when you wish to insert a new value (node) into the
            ↪  DoublyLinkedList<T>
93          private Node<T> AddBetween(T value, Node<T> previous, Node<T> next)
94          {
```

```
95              Node<T> node = new Node<T>(value, previous, next);
96              previous.Next = node;
97              next.Previous = node;
98              Count++;
99              return node;
100         }
101
102         public INode<T> Find(T value)
103         {
104             Node<T> node = Head.Next;
105             while (!node.Equals(Tail))
106             {
107                 if (node.Value.Equals(value)) return node;
108                 node = node.Next;
109             }
110             return null;
111         }
112
113         public override string ToString()
114         {
115             if (Count == 0) return "[]";
116             StringBuilder s = new StringBuilder();
117             s.Append("[");
118             int k = 0;
119             Node<T> node = Head.Next;
120             while (!node.Equals(Tail))
121             {
122                 s.Append(node.ToString());
123                 node = node.Next;
124                 if (k < Count - 1) s.Append(",");
125                 k++;
126             }
127             s.Append("]");
128             return s.ToString();
129         }
130
131         // TODO: Your task is to implement all the remaining methods.
132         // Read the instruction carefully, study the code examples from above as
            ↪ they should help you to write the rest of the code.
133
134         /// <summary>
135         /// Returns the node before the passed in node, if both exist. If the node
            ↪ does not exist, null
136         /// is returned.
137         /// </summary>
138         /// <param name="node"></param>
139         /// <returns>The node before the passed in node</returns>
140         /// <exception cref="System.ArgumentNullException">Thrown when the node is
            ↪ null</exception>
141         /// <exception cref="System.InvalidOperationException">Thrown when the node
            ↪ to
142         /// return the node of before, does not exist</exception>
143         public INode<T> Before(INode<T> node)
```

```
144         {
145             if (node is null) throw new NullReferenceException();
146             Node<T> result = node as Node<T>;
147             if (result.Previous == null || result.Next == null)
148                 throw new InvalidOperationException("The node referred as 'before'
                    ↪  is no longer in the list");
149             if (result.Previous.Equals(Head)) return null;
150             else return result.Previous;
151         }
152 
153         /// <summary>
154         /// Adds a new node at the head of the linked list
155         /// </summary>
156         /// <param name="value">The value to add as a new node</param>
157         /// <returns>The node added to the head of the list</returns>
158         public INode<T> AddFirst(T value)
159         {
160             return AddBetween(value, Head, Head.Next);
161         }
162 
163         /// <summary>
164         /// Adds a node into the linked list before an existing node
165         /// </summary>
166         /// <param name="before">The node to add before</param>
167         /// <param name="value">The value to add as a new node</param>
168         /// <returns>Node added to the list</returns>
169         /// <exception cref="System.ArgumentNullException">Thrown when the node is
            ↪  null</exception>
170         /// <exception cref="System.InvalidOperationException">Thrown when the node
            ↪  to
171         /// remove does not exist</exception>
172         public INode<T> AddBefore(INode<T> before, T value)
173         {
174             if (before is null) throw new NullReferenceException();
175             Node<T> result = before as Node<T>;
176             if (result != null || result.Next != null)
177                 return AddBetween(value, result.Previous, result);
178             throw new InvalidOperationException();
179         }
180 
181         /// <summary>
182         /// Inserts a node with the required value, after an existing now
183         /// </summary>
184         /// <param name="after">The node with value to add after</param>
185         /// <param name="value">The value to add as a new node</param>
186         /// <returns>Node added to the linked list</returns>
187         /// <exception cref="System.ArgumentNullException">Thrown when the node is
            ↪  null</exception>
188         /// <exception cref="System.InvalidOperationException">Thrown when the node
            ↪  to
189         /// remove does not exist</exception>
190         public INode<T> AddAfter(INode<T> after, T value)
191         {
```

```
192             if (after is null) throw new ArgumentNullException();
193             Node<T> result = after as Node<T>;
194             if (result.Previous != null || result.Next != null)
195                 return AddBetween(value, result, result.Next);
196             throw new InvalidOperationException();
197         }
198 
199         /// <summary>
200         /// Clear all nodes from the linked list and returns it's state
201         /// back to the same as a new linked list.
202         /// </summary>
203         public void Clear()
204         {
205             Node<T> current = Head.Next;
206             while (!current.Equals(Tail))
207             {
208                 Node<T> temp = current;
209                 current = current.Next;
210                 InvalidateNode(temp);
211             }
212             Head.Next = Tail;
213             Tail.Previous = Head;
214             Count = 0;
215         }
216 
217         /// <summary>
218         /// Removes a node from the linked list if it exists.This will remove the
            ↪ first
219         /// instance of a node with the defined value.
220         /// We need to use an O(n) approach for our existing data structure to
            ↪ prevent
221         /// the same node being removed twice, and to prevent nodes in one list
            ↪ being
222         /// used to remove nodes in another list.
223         /// Using Find ensures we know 100% that the node exists in the list we
            ↪ want to
224         /// remove it from.
225         /// </summary>
226         /// <param name="node">The node to remove if it exists. This will find only
227         /// the first occurence of the node with the corresponding value</param>
228         /// <exception cref="System.ArgumentNullException">Thrown when the node is
            ↪ null</exception>
229         /// <exception cref="System.InvalidOperationException">Thrown when the node
            ↪ to
230         /// remove does not exist</exception>
231         public void Remove(INode<T> node)
232         {
233             if (node is null) throw new ArgumentNullException();
234             Node<T> result = Find(node.Value) as Node<T>;
235             if (result is null) throw new InvalidOperationException();
236             result.Previous.Next = result.Next;
237             result.Next.Previous = result.Previous;
238             InvalidateNode(result);
```

```
239             Count--;
240         }
241 
242         private void InvalidateNode(Node<T> node)
243         {
244             node.Next = null;
245             node.Previous = null;
246         }
247 
248         /// <summary>
249         /// Removes the Head node in the linked list
250         /// </summary>
251         public void RemoveFirst()
252         {
253             if (Count is 0) throw new InvalidOperationException();
254             Node<T> node = Head.Next;
255             Head.Next = node.Next;
256             Head.Next.Previous = Head;
257             InvalidateNode(node);
258             Count--;
259         }
260 
261         /// <summary>
262         /// Removes the Tail node in the linked list
263         /// </summary>
264         public void RemoveLast()
265         {
266             if (Count is 0) throw new InvalidOperationException();
267             Node<T> node= Tail.Previous;
268             Tail.Previous = node.Previous;
269             Tail.Previous.Next = Tail;
270             InvalidateNode(node);
271             Count--;
272         }
273     }
274 }
```