**LinkedList Design Document**

Arjun Akkiraju P4

**Description**

The MyLinkedList stores objects in a doubly linked list of nodes. A node is an object which stores a value, and two pointers which point to the next node in the linked list, and the previous node in the linked list. The MyLinkedList is a sequential data structure, making it great for storing values in which there is a concept of order, that is, one object comes before another object. MyLinkedLists are dynamic data structures, meaning that they can grow and shrink to accommodate the number of nodes they need to store at runtime. They are also indexed starting from 0. MyLinkedLists also make the insertion/deletion of stored objects simpler, since, instead of shifting each object left or right like an ArrayList, MyLinkedLists simply alter the pointers of the nodes to point to a new node. Any object can be stored in a MyLinkedList, including null, and duplicate objects can be stored in MyLinkedLists as well. In order to efficiently iterate through a MyLinkedList, the class provides an iterator object which can quickly iterate over the MyLinkedLists object in one direction, and perform simple operations like adding/removing nodes, while also providing access to the next node in the MyLinkedList object.

**Services**

A brief overview of some of the methods/services the MyLinkedList class provides.

**MyLinkedList() -** creates a new MyLinkedList object. Initializes three instance fields, *first*, *last*, and *size*. The instance fields *first* and *last* point to the first and last nodes in the doubly linked list. Both are set to null. Since there has been nothing added to the MyLinkedList, *size* which keeps track of the size of the MyLinkedList object is set to 0.

**String toString() -** returns a string which contains a formatted list of the values stored in each node stored by the MyLinkedList object. Because the method uses a while loop to iterate over each object stored in the MyLinkedList, it runs in O(n) time.

**int size() -** returns the sizeof the MyLinkedList object in O(1) time. The method does this by returning the value of the *size* instance field, which keeps track of the number of nodes in the MyLinkedList.

**E get(int index) -** returns the object stored in the MyLinkedList at the specified index. The MyLinkedList object does not have random access capabilities, meaning that it must traverse through the MyLinkedList until it arrives at the specified index. Once it has done this, it must return the value of the node at the specified index. As a result, the operation runs in O(n) speed.

**E set(int index, E obj) -** sets the current node’s value at the specified index to the specified object. The method returns the value which has been replaced. Because MyLinkedLists do not have random access capabilities, they must traverse through the MyLinkedList until the specified index matches the current index before changing the value at the current node. Thus, the operation runs in O(n) time. A postcondition of the set method is that the node at the specified index will contain the specified object.

**boolean add(E obj) -** adds the specified object to the end of the MyLinkedList object. The add method modifies the *size* instance field by incrementing it by 1. The MyLinkedList method will access its *last* instance field and simply set the last node’s next node to a new node containing the specified object and pointing the null. As a result, the method runs in O(1) time. A postcondition of this method is that a new node which contains the specified object will be appended to the back of the MyLinkedList. The instance field *last* for the node previously at the last index will point to the newly appended node instead of null. The newly appended node will have its *first* instance field point to the node before it and its *last* instance field point to null. Returns true always.

**E remove(int index) -** Removes the node at the specified index and returns the value of that node. The method does this by finding the node 1 index before the specified index and setting its *last* instance field to the node 2 indices to its right and the node 1 index after the specified index sets its *first* instance field to the node 2 indices to its left. To reach these indices, the MyLinkedList must traverse through the linked list used to implement the MyLinkedList. Because of this, the method runs in O(n) time. The *size* instance field is decremented by 1.

**void add(int index, E add) -** adds the specified object to the MyLinkedList at the specified index. The add method does this by traversing through the linked list storing the data and stopping when it has reached the specified index. The method then creates a new node which stores the specified value and sets its first pointer to the node before it and its last node to the node after it. After this the method sets *last* instance field of the node before the specified index to the newly created node and sets the *first* instance field of the node after the specified node to the newly created node, maintaining the structure of the doubly linked lists. Because the method traverses through n elements, the method runs in O(n) time. The *size* instance field is incremented by 1. A postcondition of this method is that a new node will be inserted at the specified index and the size of the MyLinkedList will increase by 1.

**void addFirst(E obj) -** adds a node whose value is the specified object at the front of the MyLinkedList. The object does this by simply setting the *first* instance variable to a new DoubleNode object which contains the object and points to null, while incrementing size variable by 1. Because it is a constant time operation, the method runs in O(1) time. A postcondition of this method is that there is a new node which contains the specified object at the first index.

**void addLast(E obj) -** adds the specified object to the end of the MyLinkedList object. The add method modifies the *size* instance field by incrementing it by 1. The MyLinkedList method will access its *last* instance field and simply set the last node’s next node to a new node containing the specified object and pointing the null. As a result, the method runs in O(1) time. A postcondition of this method is that a new node which contains the specified object will be appended to the back of the MyLinkedList. The instance field *last* for the node previously at the last index will point to the newly appended node instead of null. The newly appended node will have its *first* instance field point to the node before it and its *last* instance field point to null. Returns true always.

**E getFirst() -** returns the value of the first node in the MyLinkedList. The method does this by simply accessing the *first* instance variable’s value, since *first* is the first node in the linked list used to implement the MyLinkedList. Because this is a constant time operation, the method runs in O(1) time.

**E getLast() -** returns the value of the last node in the MyLinkedList. The method does this by traversing through the entire MyLinkedList until it reaches the final node. The method then returns that node’s value. Because the method performs n operations, the method runs in O(n) time.

**E removeFirst() -** returns the value of the first node, which is being removed. To remove the first node, the *first* instance field of the MyLinkedList simply points to the node directly preceding the first node. Because this is a constant time operation, the operation runs in O(1) time. A postcondition of this method is that the first node in the MyLinkedList is second node of the MyLinkedList before the method was called. As a result, the *size* instance field is decremented by 1.

**E removeLast() -** returns the value of the last node, which is being removed. To remove the last node, the *last* instance field of the MyLinkedList points to the node preceding the last node. Because this is a constant time operation, the method runs in O(1) time. A postcondition of this method is that the last node in the MyLinkedList is second to last node of the MyLinkedList before the method was called. As a result, the *size* instance field is decremented by 1.

**Iterator<E> iterator() -** returns a new MyLinkedListIterator object which performs simple operations such as adding/removing elements, and providing access to the MyLinkedList’s next element quickly and efficiently

**Internal State and Data Structures**

The MyLinkedList implements a doubly linked list to store its data sequentially. A doubly linked list is a collection of nodes which store both the data and a pointer to the next and previous nodes. Doubly linked lists can be traversed in both directions and are great for efficiently inserting and removing data from the list. The MyLinkedList has both a *first*, *last*, and *size* instance field which specify the first node in the MyLinkedList, the last node in the MyLinkedList, and the size of the MyLinkedList respectively.

**Testing Procedure**

Below is a detailed procedure on how to test the **boolean add(int index, E obj)** method.

Execute the following java statements.

MyLinkedList<Object> x = new MyLinkedList();

x.add(4);

x.add(6);

x.add(7);

x.add(2);

System.out.println(x.toString());

This should result in a MyLinkedList with 4 values formatted like so: [4, 6, 7, 2]. Suppose we want to add another object to our MyLinkedList. Say it is a string containing “hello”. In order to test the method, make a series of calls to the boolean add(int index, E obj) method.

Execute the following statements.

String test = “hello”;

x.add(1, test);

x.add(1, test);

x.add(4, test);

System.out.println(x.toString());

The print statement should have yielded a list of 7 values formatted like so:

[4, “hello”, “hello”, 7, “hello”, 2]. If this is true, the method is working as intended. Now, if an index beyond the range of the MyLinkedList object’s length is used, the method should throw an indexOutOfBoundsException. Try executing the following statement.

x.add(100, “error”);

This should throw the indexOutOfBoundsException.

Now, try to add an element to the back of the array using the index. Suppose the MyLinkedList object still contains the following: [4, “hello”, “hello”, 7, “hello”, 2]. Try executing the following java statement.

x.add(6, “end”);

This should not throw the indexOutOfBoundsException and should instead, simply append the value “end” to the end of the MyLinkedList object.

Below are general instructions on testing the rest of the methods, code samples included.

**String toString()**

Execute the following statements

. MyLinkedList<Object> x = new MyLinkedList();

x.add(4);

x.add(6);

x.add(7);

x.add(2);

System.out.println(x.toString());

The result should be [4, 6, 7, 2].

**int size()**

Execute the following statements

. MyLinkedList<Object> x = new MyLinkedList();

x.add(4);

x.add(6);

x.add(7);

x.add(2);

System.out.println(x.size());

The result should be 4

**E get(int index)**

Execute the following statements

. MyLinkedList<Object> x = new MyLinkedList();

x.add(4);

x.add(6);

x.add(7);

x.add(2);

System.out.println(x.get(2));

The result should be 6

**E set(int index, E obj)**

Execute the following statements

. MyLinkedList<Object> x = new MyLinkedList();

x.add(4);

x.add(6);

x.add(7);

x.add(2);

x.set(0, 9);

System.out.println(x.toString());

The result should be [9, 6, 7, 2]

**boolean add(E obj)**

Execute the following statements

. MyLinkedList<Object> x = new MyLinkedList();

x.add(9);

x.add(6);

x.add(7);

x.add(2);

x.add(9);

System.out.println(x.toString());

The result should be [9, 6, 7, 2, 9]

**E remove(int index)**

Execute the following statements

. MyLinkedList<Object> x = new MyLinkedList();

x.add(9);

x.add(6);

x.add(7);

x.add(2);

x.remove(0);

System.out.println(x.toString());

The result should be [6, 7, 2]

**void add(int index, E obj)**

Execute the following statements

. MyLinkedList<Object> x = new MyLinkedList();

x.add(9);

x.add(6);

x.add(7);

x.add(2);

x.add(2, 19);

System.out.println(x.toString());

The result should be [9, 6, 19, 7, 2]

**void addFirst(E obj)**

Execute the following statements

. MyLinkedList<Object> x = new MyLinkedList();

x.add(9);

x.add(6);

x.add(7);

x.add(2);

x.addFirst(9);

System.out.println(x.toString());

The result should be [9, 9, 6, 7, 2]

**void addLast(int index, E obj)**

Execute the following statements

. MyLinkedList<Object> x = new MyLinkedList();

x.add(9);

x.add(6);

x.add(7);

x.add(2);

x.add(2, 19);

System.out.println(x.toString());

The result should be [9, 6, 19, 7, 2]

**E getFirst()**

Execute the following statements

. MyLinkedList<Object> x = new MyLinkedList();

x.add(9);

x.add(6);

x.add(7);

x.add(2);

System.out.println(x.getFirst());

The result should be 9

**E getLast()**

Execute the following statements

. MyLinkedList<Object> x = new MyLinkedList();

x.add(9);

x.add(6);

x.add(7);

x.add(2);

System.out.println(x.getLast());

The result should be 2

**E removeFirst()**

Execute the following statements

. MyLinkedList<Object> x = new MyLinkedList();

x.add(9);

x.add(6);

x.add(7);

x.add(2);

x.removeFirst();

System.out.println(x.toString());

The result should be [6,7,2]

**E removeLast()**

Execute the following statements

. MyLinkedList<Object> x = new MyLinkedList();

x.add(9);

x.add(6);

x.add(7);

x.add(2);

x.removeLast();

System.out.println(x.toString());

The result should be [9, 6, 7]

**Iterator<E> iterator()**

**E removeFirst()**

Execute the following statements

. MyLinkedList<Object> x = new MyLinkedList();

Y = x.iterator();

Y should be a new MyLinkedListIterator object.