**MyArrayList Design Document**

Arjun Akkiraju P4

**Description**

The MyArrayList stores objects in a list and allows for those objects to be accessed at any time. This is great for storing a list of values in which there is a concept of order, that is, one object comes before another object. Any object can be stored in the MyArrayList, including null. The MyArrayList object is expandable, meaning that its capacity can expand and shrink depending on how many objects are being stored. MyArrayList objects are indexed starting from 0 and allows for duplicates within the list. In order to efficiently iterate through a MyArrayList object, the class provides an iterator object which can quickly iterate over the MyArrayList object and perform simple operations like adding/removing elements, while also providing access to the next element in the MyArrayList object.

**Services**

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

**MyArrayList() -** creates a new MyArrayList object. Initializes two instance fields, *size* and *values*. The instance field *size* stores the number of elements being stored in MyArrayList object and the *values* instance field is an array which is used to store the values being stored in the MyArrayList object.

**int size()**- returns the size of the MyArrayList 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 elements in the MyArrayList object.

**E get(int index)**- returns the object stored in the MyArrayList at the specified index. MyArrayList objects have random access capabilities, meaning that the operation is performed in O(1) time.

**E set(int index, E obj)** - sets the current object at the specified index to a the specified object. The method returns the element previously at the specified index. Because the set method needs to shift the elements to the right of the specified index to the right, it performs n operations, meaning the method runs in O(n) time. A postcondition of the set method is that the element at the specified index will be the specified object. As a result, the instance field *values*, which is the array which keeps track of the elements stored in the MyArrayList, is modified to include the specified object at the specified index.

**E remove(int index)** - removes the object at the specified index and shifts any following elements in the list to the left by subtracting from 1 from their index. Because the MyArrayList object needs to shift each subsequent element to the left by 1, the MyArrayList object performs the remove method in O(n) time. The method returns the element which was removed from the MyArrayList object. A postcondition of the method is that the element a the specified index is removed, and all subsequent elements in the MyArrayList objects to the left by 1 index. As a result, the aforementioned *size* instance field is reduced by 1, and the *values* instance field by removing the element at the specified index.

**boolean add(E obj)** - adds the specified object to the end of the MyArrayList object. The add method modifies the aforementioned *values* instance field by adding the specified object to the end of the *values* array. If the *values* array has no space left and is at maximum capacity, the *values* array is doubled to accomodate for the object. If the current array *values* has enough capacity to store one more object, the operation runs at O(1) time. If not, the *values* array needs to double its capacity, which means it needs to add n spaces to the back of the array.Thus, if the *values* array is at full capacity, the operation will be performed in O(n) time. The *size* instance field is incremented by 1. A postcondition of the add method is that the specified object will be appended to the end of the MyArrayList object.

**boolean add(int index, E obj)** - adds the specified object to the MyArrayList object at the specified index. The add method modifies the *values* array by inserting the specified object to the array at the specified index and shifts all subsequent elements to the right by 1 index. Because of this, the method needs to shift approximately every element to the right by 1 space, meaning that the operation runs at O(n) time. If the object is added and the *values* array is full, the capacity of the *values* array would double and 2n operations would be performed (n operations to shift the subsequent elements to the right, and n operations to double the capacity). 2n rounds down to n in big O notation, the method runs in O(n) time. The *size* instance field is incremented by 1. A postcondition of the add method is that the object will be added to the MyArrayList object at the specified index, with each subsequent element being shifted down by 1 index.

**void doubleCapacity()** - doubles the capacity (number of elements the array can store) of the *values* array. Because the method needs to double the amount of elements that the array can store, the operation runs in O(n) time, since it needs to perform n operations to double the length of the *values* array. A postcondition is that the *values* array is doubled in length.

**int getCapacity()**- returns the length of the *values* array. Because of the MyArrayList’s random access capabilities, the operation runs in O(1) time.

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

**String toString()** - returns a string which contains a formatted list of the elements stored in the MyArrayList object. Because the method uses a for loop to iterate over the array being used to store the elements, the operation runs at O(n) time.

**Internal Data Structures and State**

The MyArrayList object uses an array to store its objects. The array is initialized in the constructor and two instance fields are created, size, which is 0, and values, which is the new Object array. Whenever the array becomes full with values, the MyArrayList object doubles its capacity so it can store more objects, since the MyArrayList object is to expandable.

**Testing Procedures**

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

Execute the following java statements.

MyArrayList<Object> x = new MyArrayList();

x.add(4);

x.add(6);

x.add(7);

x.add(2);

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

This should result in a MyArrayList with 4 values formatted like so: [4, 6, 7, 2]. Suppose we want to add another object to our MyArrayList. 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 MyArrayList 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 MyArrayList 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 MyArrayList object.