CMSC 206: Data Structures
Lab 7: Linked Lists & Iterators

For this lab, you will be writing an iterator for the SingleLinkedList class, as adapted from our textbook. You will then use this iterator to conduct a few performance experiments comparing KWArrayList and SingleLinkedList.

Part I.

1. Make a new project for this lab. Copy the SingleLinkedList and KWArrayList classes into it.

2. In the SingleLinkedList class, make an private SLLIterator class. Here is how to start it (the bold part is what you have to write; the rest is just for context):

   ```java
   public class SingleLinkedList<E>
   {
       private static class Node<E> { ... }

       private static class SLLIterator<E>
           implements Iterator<E>
       {
           ...
       }

       private Node<E> head = null;
       ...
   }
   
   Although in homework, exams, etc., you won't have to write classes inside of others, we need to do so here in order to access the Node class within SingleLinkedList.

3. The definition above will not compile, because the concrete SLLIterator inherits abstract methods from Iterator. Look up the Iterator documentation. (It's in java.util.) You will have to override the methods hasNext and next to get this to work.

4. Your iterator will need to store information to keep track of where in the list it is. For an array, storing an index is appropriate, but doing so for a linked list would be inefficient (every access of the list would require walking down the list from the head). So, we will use a Node field in the SLLIterator class:

   ```java
   private Node<E> node;
   ```
5. Write a constructor as follows:

```java
/** Creates this iterator at the beginning of the
 * given list
 * @param list The list to iterate down
 */
private SLLIterator(SingleLinkedList<E> list)
{
    // you write the code here
}
```

This must initialize the one field of your `SLLIterator` class to refer to the head of the provided list. It's only one line of code!

6. The `node` field of `SLLIterator` will refer to the node whose data will next be returned by the `next` method.

![Diagram](image)

In the scenario pictured here, `hasNext()` should return `true`, and `next()` should return "b" while advancing the `node` to refer to the node containing "c".

What will `node` be when it's at the end of the list (i.e., there are no more elements to return)?

7. Write the `hasNext()` and `next()` methods for `SLLIterator`. Make sure that the `next()` method throws a new `NoSuchElementException` if `next()` is called when there are no more elements. (If you don't handle this explicitly, your method will likely throw a `NullPointerException` in the no-more-elements case.)

8. Now, you need to make it possible for a user to obtain this iterator. Add the following method to the `SingleLinkedList` class (not the `SLLIterator` class!):

```java
/** @return an iterator over this list */
public Iterator<E> iterator()
```

This method should be about one line, and it will use the keyword `this`. Note that the return type is `Iterator<E>`, even though you will return a `SLLIterator<E>` object. This is OK because `SLLIterator` inherits from `Iterator`. 
9. In order for the SingleLinkedList class to work with the "foreach" loop, it needs to implement the Iterable interface. Modify SingleLinkedList to implement this interface by changing the header line to look like this:

```java
public class SingleLinkedList<E>
    implements Iterable<E>
```

You shouldn't need to make any more changes, because the Iterable interface defines only the iterator method, which you've already written.

10. Make a separate Main class, with the following main method:

```java
public class Main
{
    public static void main(String[] args)
    {
        SingleLinkedList<String> list =
            new SingleLinkedList<>();

        list.addFirst("snow");
        list.addFirst("it");
        list.addFirst("let");

        for(String s : list)
        {
            System.out.println(s);
        }
    }
}
```

This uses the fact that the type of list is something that implements Iterable to call the iterator method and then uses hasNext and next to iterate through the list.

Run this method and confirm it works.

Why does it use addFirst to build up the list instead of add?

11. Add System.out.println statements to iterator, hasNext, and next to demonstrate that the "foreach" loop really does call these methods.
Part II.

Similar to how we did so in class, you will measure the performance of the two collections you have.

12. Add a field to the SingleLinkedList class to track how many times the node = node.next line in getNode is called. That field should start out at 0 and be incremented every time the loop in getNode is run. (Note that this loop is the only loop in SingleLinkedList.) You will also have to add a getter for this field so that a main method can print it.

13. Write a new main method (in a new class) that builds a list by successively adding new elements to the end. Have it build lists of length 10, 20, 40, and 80. For each list, print out how many times the node = node.next line in getNode was called. How does this growth depend on the length of the list?

14. Change your main method to use addFirst instead of add. (This will build the lists in reverse order.) How many times does the loop get run now?

15. Change your main method to use KWArrayList instead of SingleLinkedList. Find the loops in KWArrayList and track how many times they run. You'll also have to change addFirst to use the two-parameter add method of KWArrayList. How many times do the loops run now?