Challenges

Challenge 1 - Implement a Linked List

Implement a simple linked list in Java without using any collections classes. The list should be implemented using a single class such that each instance represents a single node in the list, encapsulating the node's value and a reference to the following node, as well as a convenience method to initialize a whole list from an array of values. The class should implement the following interface:

```
public interface LinkedListNode<E> {

    /* getter/setter for this node's value */
    E getValue();
    void setValue(E value);

    /* getter/setter for the subsequent node, or null if this is the last node */
    LinkedListNode<E> getNext();
    void setNext(LinkedListNode<E> next);

    /**
     * Initialize this node and all of its subsequent nodes from
     * an array of values, starting with the head value at index 0
     *
     * @param listValues - the ordered values for the whole list
     */
    void setValuesFromArray(E[] listValues);

}
```

When complete, you should be able to successfully run the following unit test using your implementation:

```
import static org.junit.Assert.*;
import org.junit.Test;
```

```java
public class LinkedListNodeTest {

    @Test
    public void test() {
        LinkedListNode<Integer> head = null;
        Integer[] listValues = new Integer[] {1, 2, 3};

        head = new LinkedListNodeImpl<>(); // replace with your implementation
        head.setValuesFromArray(listValues);

        assertEquals(listValues[0], head.getValue());
        assertNotNull(head.getNext());
        assertEquals(listValues[1], head.getNext().getValue());
        assertNotNull(head.getNext().getNext());
        assertEquals(listValues[2], head.getNext().getNext().getValue());
        assertNull(head.getNext().getNext().getNext());
    }

}
```

Challenge 2 - Add two numbers

In this exercise, we'll use your `LinkedListNode` implementation to represent a non-negative integer such that each node in the list represents a single digit (in base 10) and the digits are stored in *reverse* order.

```
1     == 1
1→2   == 21
1→2→3 == 321
```

Write a program which takes as its input two such lists, *a* and *b*, and adds them arithmetically one decimal at a time. Your algorithm should traverse both lists together, adding the values for each node and carrying the 1 to the next place when the sum >= 10. The result should be returned as a linked list in the same format as the input lists.

Examples:

```
1→2     +   5→3   == 6→5      // 21 + 35 == 56
1→2     +   1→2→3 == 2→4→3    // 21 + 321 == 342
1→2→3   +   7→8→9 == 8→0→3→1  // 321 + 987 == 1308
```

Challenge 3 - Evaluating Time Complexity

Identify the time and space complexity of the following using Big O notation:

- What is the time and space complexity of the `LinkedListNode.setValuesFromArray` method in Challenge #1?
- What is the time and space complexity of addition algorithm in Challenge #2?

Hint: the Big O complexity for the addition algorithm will require two variables to reflect the fact that two inputs are used

Challenge 4 - Hash table word count

Write a program which takes as its input a `String` of natural language text and outputs a`HashMap<String,Integer>` whose keys are the unique words in the input and whose values are the number of times those words occur. The algorithm should be *case-insensitive* (e.g. "Program" and "program" would count as the same word) and ignore punctuation and whitespace.

Example: Given the input `"To be or not to be, that is the question"`, the outputted `HashMap`would contain 8 entries, with two words having a count of 2 and six words having a count of 1:

```
"to"        → 2
"be"        → 2
"or"        → 1
"not"       → 1
"that"      → 1
"is"        → 1
"the"       → 1
"question"  → 1
```

Challenge 5 - Multimaps

One common pattern when using hash tables requires building a `Map` whose values are `Collection`instances. In this challenge, we'll take the output of the previous challenge and invert it.

Write a program that takes as its input a `HashMap<String,Integer>` and returns a`HashMap<Integer,HashSet<String>>` containing the same data as the input map, only inverted, such that the input map's values are the output map's keys and the input map's keys are the output map's values.

Example:

Consider the example output for Challenge #4. Using that map as the input, the output for this challenge would be:

```
2 → ["to", "be"]
1 → ["or", "not", "that", "is", "the", "question"]
```

Challenge 6 - Longest non-repeating substring

Write a program which takes as its input a `String` and returns the length of the longest substring that does *not* contain any repeated characters.

Example: Given the string `"abcabcbb"`, the longest substring with no repeated characters is `"abc"`, so the program would return a value of `3`. Given the string `"aaaaaaaa"`, the longest non-repeating substring is `"a"` and thus the output would be `1`.

Interview Questions

Interview Question 1 - Linked List Duplicate Removal

Write a program which takes as its input an unsorted linked list of integers and deletes any duplicate values from the list *without using a temporary buffer collection* or any additional collection classed, such as a`HashSet`. Use the `LinkedListNode` implementation from this week's challenge.

Example: Given a linked list `1→2→3→3→4→4`, the program should output: `1→2→3→4`

Once you've implemented the algorithm, identify the *time and space complexity* of your solution using Big O notation.

Bonus 1

Implement an alternative solution, this time using a temporary storage buffer. This second solution should represent a significant improvement over the first in terms of time complexity, though it will come at the cost of additional space complexity. Be sure you can identify both using Big O notation.

Hint: Refer to the Big O Notation page and consult the the noted examples to determine which of them is the best fit.

Interview Question 2 - Implement a Hashtable

Implement a simple hashtable without using any special collections classes or helpers (you can use native Java arrays). Your implementation should minimally meet the following requirements:

- Support generics for both key and value types
- Implement the standard `get`, `put`, `remove`, `size`, `clear`, and `isEmpty` operations as defined in `java.util.Map`
- Support an initial default capacity of 16 entries
- Support dynamic allocation of additional capacity as needed

Bonus 1

Implement the entire `java.util.Map` interface by adding support for the remaining operations:

- Implement `containsKey` and `containsValue`
- Implement `keySet`, `keySet`, and `values`
- Implement `putAll`

Hint: One standard approach to this problem would involve using two-dimensional arrays.