GetAhead - Interview Practice 1

Flattened Iterator - Solution

Given a list of iterators, implement a `FlattenedIterator` class which incrementally iterates over the integers from all the iterators in an interleaved fashion.

Example:

```
Iterators[0] = [1,2,3]
Iterators[1] = [4,5]
Iterators[2] = [6,7,8]

FlattenedIterator = [1, 4, 6, 2, 5, 7, 3, 8]
```

An iterator implements the `next()` and `hasNext()` interface. You're free to use them, and you will implement them on the `FlattenedIterator` class.

You're free to initialize `FlattenedIterator` with any data structure of your choice for the iterators.

Solution

There are two possible solutions with different time complexity properties. The first solution keeps the index of the iterator it will return its next value from, which involves having to search for the next non-empty iterator. Creating the `FlattenedIterator` takes O(1) time but `next()` and `hasNext()` take O(N), where N is the size of the `Iterator` array.

The second solution, which we will describe here in more detail, instead keeps a queue of remaining non-empty iterators. Iterators are added to the queue in the constructor, in the provided order and only if they are non-empty. The time complexity of the constructor is therefore O(N) because it needs to check every iterator's emptiness.

On the other hand, the operations have become much simpler and both take constant time. `hasNext()` returns true if and only if the queue is not empty (because only iterators with values are kept in the queue). `next()` removes the first iterator from the front of the queue, returns its integer value and then puts the iterator back at the end of the queue if it has more values. If the iterator is now empty, it is dropped. Because of the way a queue works, we will first return integers from all other remaining non-empty iterators before coming round to this one again, hence returning integers in an interleaved fashion.

So which solution is better? With the index-based solution, creating a `FlattenedIterator` takes constant time but invoking its operations takes linear time. Conversely, the queue-based solution requires linear time to create the `FlattenedIterator` but its operations take constant time. A student can ask the interviewer which will be dominant but the most likely answer is that there are more values in the iterators than there are iterators. The student could therefore state this as an assumption in the beginning and prioritize cheap `next`/`hasNext` operations over the construction time.

```java
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

public class FlattenedIterator implements Iterator<Integer> {
   public FlattenedIterator(List<Iterator<Integer>> iterators) {
       // Create queue of iterators. We choose linked list as its backing
       // data structure because both add() and remove() take O(1) time.
       mIteratorQueue = new LinkedList<>();

       // Go over iterators in the provided order and add them to the
       // iterator queue if  they are not empty. This loop takes O(N) time,
       // where N is the length of iterators.
       for (Iterator<Integer> iterator : iterators) {
           if (iterator.hasNext()) {
               mIteratorQueue.add(iterator);
           }
       }
   }

   @Override
   public boolean hasNext() {
       // Check if iterator queue is empty. If it is, there are no more
       // values available.
       if (mIteratorQueue.isEmpty()) {
           return false;
       }

       // Optional: Check that the iterator in the front of the queue has a
```

```
        // value available.
        // We never add an empty iterator to the queue in our code, so this only:
        // (a) serves as a sanity check that we have not made a mistake in our
        // code, and
        // (b) checks that the iterator was not modified externally.
        // Throw an exception if the check fails.
        if (!mIteratorQueue.peek().hasNext()) {
            throw new IllegalStateException("Iterator unexpectedly empty");
        }

        // There is a next value available in the iterator in the front of
        // the queue.
        return true;
    }

    @Override
    public Integer next() {
        // Call hasNext() to check whether a value is available in the
        // first iterator in the queue.
        if (hasNext()) {
            // Remove the first iterator from the queue. Obtain its next value
            // and return the iterator back to the queue if it has more values
            // left in it.
            Iterator<Integer> iterator = mIteratorQueue.remove();
            Integer value = iterator.next();
            if (iterator.hasNext()) {
                mIteratorQueue.add(iterator);
            }
            return value;
        } else {
            // No value available, return null.
            return null;
        }
    }

    private Queue<Iterator<Integer>> mIteratorQueue;
}
```

A good solution should also include tests. Here we would check the order of returned values for a small number of iterators with varying lengths. We must also cover corner cases like creating `FlattenedIterator` with an empty `Iterators` array, or an array containing empty iterators. Although you are not expected to write formal test code in an interview, it is important that you list what test cases you would use, and that you try manually your code with a few basic ones.

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

public class FlattenedIteratorTest {
   @Test
   public void testNoSubIterators() {
       FlattenedIterator it = new FlattenedIterator(List.of());
       assertNull(it.next());
       assertFalse(it.hasNext());
   }

   @Test
   public void testEmptySubIterators() {
       List<Integer> listA = List.of();
       List<Integer> listB = List.of();
       FlattenedIterator it = new FlattenedIterator(
           List.of(listA.iterator(), listB.iterator()));
       assertNull(it.next());
       assertFalse(it.hasNext());
   }

   @Test
   public void testSingleSubIterator() {
       List<Integer> listA = List.of(1, 2);
       FlattenedIterator it = new FlattenedIterator(List.of(listA.iterator()));
       assertTrue(it.hasNext());
       assertEquals(Integer.valueOf(1), it.next());
       assertEquals(Integer.valueOf(2), it.next());
       assertNull(it.next());
       assertFalse(it.hasNext());
   }

   @Test
   public void testSameLengthSubIterators() {
       List<Integer> listA = List.of(1);
       List<Integer> listB = List.of(2);
       FlattenedIterator it = new FlattenedIterator(
           List.of(listA.iterator(), listB.iterator()));
       assertTrue(it.hasNext());
       assertEquals(Integer.valueOf(1), it.next());
       assertEquals(Integer.valueOf(2), it.next());
       assertNull(it.next());
       assertFalse(it.hasNext());
   }
```

```java
    @Test
    public void testDifferentLengthSubIterators() {
        List<Integer> listA = List.of(1, 4);
        List<Integer> listB = List.of(2);
        List<Integer> listC = List.of();
        List<Integer> listD = List.of(3, 5, 6);
        FlattenedIterator it = new FlattenedIterator(
            List.of(
                listA.iterator(), listB.iterator(),
                listC.iterator(), listD.iterator()));
        assertTrue(it.hasNext());
        assertEquals(Integer.valueOf(1), it.next());
        assertEquals(Integer.valueOf(2), it.next());
        assertEquals(Integer.valueOf(3), it.next());
        assertEquals(Integer.valueOf(4), it.next());
        assertEquals(Integer.valueOf(5), it.next());
        assertEquals(Integer.valueOf(6), it.next());
        assertNull(it.next());
        assertFalse(it.hasNext());
    }
}
```