Iterator of Iterators

Here is a video walkthrough of the solutions.

Implement an `IteratorOfIterators` which will accept as an argument a `List` of `Iterator` objects containing `Integers`. The first call to `next()` should return the first item from the first iterator in the list. The second call to `next()` should return the first item from the second iterator in the list. If the list contained `n` iterators, the `n+1`th time that we call `next()`, we would return the second item of the first iterator in the list.

Note that if an iterator is empty in this process, we continue to the next iterator. Then, once all the iterators are empty, `hasNext` should return **`false`**. For example, if we had 3 `Iterators` A, B, and C such that A contained the values `[1, 3, 4, 5]`, B was empty, and C contained the values `[2]`, calls to `next()` for our `IteratorOfIterators` would return `[1, 2, 3, 4, 5]`.

```
1  import java.util.*;
2  public class IteratorOfIterators  _______________________________ {
3
4
5      public IteratorOfIterators(List<Iterator<Integer>> a) {
6
7
8
9
10
11
12
13     }
14
15     @Override
16     public boolean hasNext() {
17
18
19
20
21     }
22
23
24
25     @Override
26     public Integer next() {
27
28
29
30
31     }
```

```
}
```

Solution: Here is a video walkthrough of the solution.

```
public class IteratorOfIterators implements Iterator<Integer> {
    LinkedList<Iterator<Integer>> iterators;

    public IteratorOfIterators(List<Iterator<Integer>> a) {
        iterators = new LinkedList<>();
        for (Iterator<Integer> iterator : a) {
            if (iterator.hasNext()) {
                iterators.add(iterator);
            }
        }
    }

    @Override
    public boolean hasNext() {
        return !iterators.isEmpty();
    }

    @Override
    public Integer next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        Iterator<Integer> iterator = iterators.removeFirst();
        int ans = iterator.next();
        if (iterator.hasNext()) {
            iterators.addLast(iterator);
        }
        return ans;
    }
}
```

Explanation: In the constructor, we make sure the iterator is not empty and add it to our list of possible iterators. For `hasNext`, we make sure that there is an iterator for us to use.

For `next`, we first make sure that there is a possible next element. If so, we get the next element from the current iterator by removing the front of our list. If the iterator still has elements left, we put it back on the end of the list for future iterations.

Alternate Solution: Although this solution provides the right functionality, it is not as efficient as the first one.

```
public class IteratorOfIterators implements Iterator<Integer> {
    LinkedList<Integer> l;

    public IteratorOfIterators(List<Iterator<Integer>> a) {
        l = new LinkedList<>();
        while (!a.isEmpty()) {
            Iterator<Integer> curr = a.remove(0);
            if (curr.hasNext()) {
                l.add(curr.next());
                a.add(curr);
            }
        }
    }

    @Override
    public boolean hasNext() {
        return !l.isEmpty();
    }

    @Override
    public Integer next() {
        if(!hasNext()) {
            throw new NoSuchElementException();
        }
        return l.removeFirst();
    }
}
```

Explanation: This solution is essentially the same as the first, except we preprocess all the elements from all iterators before going into `hasNext` or `next`. This is less efficient because we may not need all these elements; for example, what if there are a million elements but our iterator is only called twice?