CS 61B Fall 2023

Iterators, Polymorphism

Exam-Level 5: September 18, 2023

1 Take Us to Your "Yrnqre"

You're a traveler who just landed on another planet. Luckily, the aliens there use the same alphabet as the English language, but in a different order.

Given the `AlienAlphabet` class below, fill in `AlienComparator` class so that it compares strings lexicographically, based on the order passed into the `AlienAlphabet` constructor. For simplicity, you may assume all words passed into `AlienComparator` have letters present in `order`.

For example, if the alien alphabet has the order `"dba..."`, which means that `d` is the first letter, `b` is the second letter, etc., then `AlienComparator.compare("dab", "bad")` should return a negative value, since `dab` comes before `bad`.

If one word is an exact prefix of another, the longer word comes later. For example, `"bad"` comes before `"badly"`. *Hint:* `indexOf` *might be helpful.*

```
1  public class AlienAlphabet {
2      private String order;
3      public AlienAlphabet(String alphabetOrder) {
4          order = alphabetOrder;
5      }
6      public class AlienComparator implements Comparator<____________> {
7          public int compare(String word1, String word2) {
8
9              int minLength = Math.min(_________________________, _________________________);
10
11             for (_____________________________________________________________________) {
12
13                 int char1Rank = _____________________________________________;
14
15                 int char2Rank = _____________________________________________;
16
17                 if (_____________________________________________) {
18                     return -1;
19
20                 } else if (_____________________________________________) {
21                     return 1;
22                 }
23             }
24
25             return _________________________________ - _________________________________;
26         }
27     }
28 }
```

Solution:

```
public class AlienAlphabet {
    private String order;

    public AlienAlphabet(String alphabetOrder) {
        order = alphabetOrder;
    }

    public class AlienComparator implements Comparator<String> {
        public int compare(String word1, String word2) {
            int minLength = Math.min(word1.length(), word2.length());
            for (int i = 0; i < minLength; i++) {
                int char1Rank = order.indexOf(word1.charAt(i));
                int char2Rank = order.indexOf(word2.charAt(i));
                if (char1Rank < char2Rank) {
                    return -1;
                } else if (char1Rank > char2Rank) {
                    return 1;
                }
            }

            return word1.length() - word2.length();
        }
    }
}
```

2 Iterator of Iterators

Implement an `IteratorOfIterators` which takes in a `List` of `Iterators` of `Integers` as an argument . The first call to `next()` should return the first item from the first iterator in the list. The second call 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]`.

```
import java.util.*;
public class IteratorOfIterators  _____________________________ {


    public IteratorOfIterators(List<Iterator<Integer>> a) {









    }

    @Override
    public boolean hasNext() {









    }

    @Override
    public Integer next() {









    }
}
```

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?