Prepared by Linan Qiu <lq2137@columbia.edu>

List Maps

Normally, computer scientists don't even speak of "list maps" since they are so inefficient that we don't even bother with them. However, I use them to illustrate the idea of **map as an Abstract Data Type**: that it can be implemented using many other data structures.

Essentially, a list map is the dumbest possible implementation of a map: it stores all the `Pair`s in a list. I can hear you screaming about efficiency already. Calm down. The main point of this example is to show you how the `Pair` class figures into maps.

List Map

```
//ListMap.java

import java.util.LinkedList;

public class ListMap<K extends Comparable<? super K>, V> implements Map<K, V> {
  private LinkedList<Pair<K, V>> list;

  public ListMap() {
    list = new LinkedList<Pair<K, V>>();
  }

  @Override
  public void put(K key, V value) {

  }

  @Override
  public V get(K key) {

  }
}
```

We create an empty `ListMap` class that simply instantiates a new linked list within its constructor. It implements the `Map` interface. Here's the important part: **the linked list holds on to `Pair`s**. Read that again and let it sink in. The dictionary always holds on to `Pair`s. Now since the linked list holds on to `Pair`s, and in `Pair`, we stipulated that `K` must be comparable, that restriction carries over to the generic definition in `ListMap`. Hence, the class declaration

reads: `public class ListMap<K extends Comparable<? super K>, V>` with the extra interface implementing bit at the end.

This solves a convenient problem: in maps, we are always dealing with a pair of variables. A linked list cannot hold two variables. For example, we cannot make a linked list hold **both a word and its definition** in the form of, say, `LinkedList<String, String> list = new LinkedList<>();`. Most data structures are designed to hold a single type. This is why `Pair` is so useful – it encapsulates the `key` and the `value` into a single class.

`put()`

Remember that `put(K key, V value)` should:

- Add a new `Pair` consisting of the given `key` and `value` if the `key` does not currently exist
- Replace the existing `value` associated with `key` if `key` already exists

Doing so with a linked list is rather straightforward, albeit inefficient. We simply iterate through the entire list. If we find a `key` that matches the `key` given, we overwrite the `value`. Otherewrise, if we have finished iterating the entire list and we still have not found any matching `key`, we can be certain that the `key` does not exist. Hence, we can add the `key` and `value` to the linked list by instantiating a new `Pair`.

```java
// ListMap.java

import java.util.LinkedList;

public class ListMap<K extends Comparable<? super K>, V> implements Map<K, V> {
  private LinkedList<Pair<K, V>> list;

  public ListMap() {
    list = new LinkedList<Pair<K, V>>();
  }

  @Override
  public void put(K key, V value) {
    for (Pair<K, V> pair : list) {
      if (pair.key.equals(key)) {
        pair.value = value;
        // we don't need to keep looking anymore.
        // we simply exit the method by returning
        // since the method is void, we don't need to
        // return anything
        return;
      }
```

```
    }
    list.add(new Pair<K, V>(key, value));
  }

  @Override
  public V get(K key) {

  }
}
```

`get()`

For `get()`, we return the `value` associated with the `key` given if the `key` exists. Otherwise, we return `null`.

```
// ListMap.java

import java.util.LinkedList;

public class ListMap<K extends Comparable<? super K>, V> implements Map<K, V> {
  private LinkedList<Pair<K, V>> list;

  public ListMap() {
    list = new LinkedList<Pair<K, V>>();
  }

  @Override
  public void put(K key, V value) {
    for (Pair<K, V> pair : list) {
      if (pair.key.equals(key)) {
        pair.value = value;
        return;
      }
    }
    list.add(new Pair<K, V>(key, value));
  }

  @Override
  public V get(K key) {
    for (Pair<K, V> pair : list) {
      if (pair.key.equals(key)) {
        return pair.value;
      }
    }
    return null;
```

```
  }
}
```

We have a working dictionary! It is inefficient, since every single operation is $O(N)$. However, it does work like a dictionary and shows you how we treat `Pair`s within the dictionary.