Prepared by Linan Qiu <lq2137@columbia.edu>

Tree Map

Can we do better than using a linked list? Sure! We can basically replace the linked list with a binary tree to get `put()` and `get()` down to $O(\log n)$ *if we have a balanced tree*. To ensure that balancing happens, we can use an **Avl Tree**. Here, we use the Weiss code for `AvlTree` modified for the following:

- Ensuring that duplicate elements added are not ignored. Instead, they overwrite old elements.
- Adding a `get()` method to `AvlTree` that, like `contains()`, searches if the element exists. However, instead of simply returning `true` or `false`, it returns the value itself.

Here's where `TreeMap` gets interesting:

1. Remember that `AvlTree` decides where to put an element added using the `compareTo` method of the element. Hence, `AvlTree` insists that anything added to it must be comparable
2. Our `Pair` is comparable. However, the `compareTo()` function in `Pair` only cares about the `key`, and not the `value`
3. Hence, to `AvlTree`, two `Pair`s with the same `key` but different `value`s will be treated the same. In other words, our dictionary treats entries with the same words but different definitions as the same. This is surprisingly conveinent, because…
4. We can overwrite `Pair`s easily by simply creating a new `Pair` that contains the same `key` but different `values`, and our `AvlTree` will (stupidly) overwrite the old `Pair` with the new `Pair` that contains new `values`. `AvlTree` won't even "notice" the difference
5. We can create dummy `Pair`s that contain only a `key` and `null` for `value`. `AvlTree` will still treat them the same as normal `Pair`s with a `value`. We can use that to search for `value`s in the `AvlTree`.

Here's the implementation.

TreeMap

```java
// TreeMap.java

public class TreeMap<K extends Comparable<? super K>, V> implements Map<K, V> {
  private AvlTree<Pair<K, V>> tree;

  public TreeMap() {
    tree = new AvlTree<Pair<K, V>>();
```

```
  }

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

  }

  @Override
  public V get(K key) {

  }
}
```

In the constructor, we instantiate a new `AvlTree`. `AvlTree` holds on to `Pairs`. Again, since we are dealing with `Pairs`, we require that `K extends Comparable<? super K`.

`put()`

```
// TreeMap.java

public class TreeMap<K extends Comparable<? super K>, V> implements Map<K, V> {
  private AvlTree<Pair<K, V>> tree;

  public TreeMap() {
    tree = new AvlTree<Pair<K, V>>();
  }

  @Override
  public void put(K key, V value) {
    tree.insert(new Pair<K, V>(key, value));
  }

  @Override
  public V get(K key) {

  }
}
```

`put()` simply creates a new `Pair` and adds it to the `AvlTree`. The add / overwrite choice is handled by the `AvlTree`, since we modified it to overwrite existing elements instead of ignoring duplicate elements. Here's the `AvlTree`'s `insert()` method that we are calling from `TreeMap`

```
// in AvlTree.java
```

```java
public void insert(AnyType x) {
  root = insert(x, root);
}

private AvlNode<AnyType> insert(AnyType x, AvlNode<AnyType> t) {
  if (t == null)
    return new AvlNode<>(x, null, null);

  int compareResult = x.compareTo(t.element);

  if (compareResult < 0) {
    t.left = insert(x, t.left);
  } else if (compareResult > 0) {
    t.right = insert(x, t.right);
  } else {
    // modified from weiss: duplicates overwrite
    t.element = x;
  }

  return balance(t);
}
```

Now since `Pair` only compares the `key` variable and not the `value`, `Pairs` with the same `key` would be treated as the "same" by `AvlTree`. Hence, the overwrite happens in the `AvlTree`.

`get()`

We make use of the trick mentioned earlier to implement `get()`:

> 5. We can create dummy `Pairs` that contain only a `key` and `null` for `value`. `AvlTree` will still treat them the same as normal `Pairs` with a `value`. We can use that to search for `values` in the `AvlTree`.

```java
// TreeMap.java

public class TreeMap<K extends Comparable<? super K>, V> implements Map<K, V> {
  private AvlTree<Pair<K, V>> tree;

  public TreeMap() {
    tree = new AvlTree<Pair<K, V>>();
  }

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

```
    tree.insert(new Pair<K, V>(key, value));
  }

  @Override
  public V get(K key) {
    Pair<K, V> found = tree.get(new Pair<K, V>(key, null));
    return found == null ? null : found.value;
  }
}
```

We create a "fake" `Pair` that consists only of the `key` given and a `null` for `value`. `get()` from `AvlTree` returns us the `Pair` that matches the `key` of the `Pair` (again owing to how we defined `compareTo` in `Pair`). The `get()` method in `AvlTree` is reproduced here:

```
// in AvlTree.java

public AnyType get(AnyType x) {
  return get(x, root);
}

private AnyType get(AnyType x, AvlNode<AnyType> t) {
  while (t != null) {
    int compareResult = x.compareTo(t.element);

    if (compareResult < 0)
      t = t.left;
    else if (compareResult > 0)
      t = t.right;
    else
      return t.element; // Match
  }

  return null; // No match
}
```

This gives us a much faster map.