1 Assorted ADTs

Below are some sketches of ADTs (not real Java code). It's not important to understand the details of how these work right now; just try to understand how each one can be used conceptually.

```
List {
    insert(item, position);      // inserts item into the list at the position
    get(position);              // returns the item in the list at the position
    size();                        // returns the number of items in the list
}

Set {
    add(item);                 // puts item in the set. Does not add duplicates
    contains(item);            // returns whether or not the item is in the set
    items();             // returns a List of all items in some arbitrary order
}

Stack {
    push(item);                                    // puts item onto the stack
    pop();                    // removes and returns the most recently put item
    isEmpty();                              // returns whether the stack is empty
}

Queue {
    enqueue(item);                                 // puts item into the queue
    dequeue();               // removes and returns the least recently put item
    isEmpty();                              // returns whether the queue is empty
}

PriorityQueue {
    enqueue(item, priority);       // puts item into the queue with a priority
    dequeue();            // removes and returns the item with highest priority
    peek();      // returns but does not remove the item with highest priority
}

Map {                                          // like a dictionary from python
    put(key, value);      /* puts key into the map and associates it with the
       given value. If key is already in the map, replaces its existing
       value with the given value */
    get(key);                                 // returns value associated with key
    keys();              // returns a List of all keys in some arbitrary order
}
```

2 Solving Problems with ADTs

Consider the problems below. Which of the ADTs given in the previous section might you use to solve each problem? Although in principle any of the ADTs might be used to solve any of the problems, think about which ones will make code implementation easier or more efficient.

1. Given a sequence of parentheses, determine whether or not they are nested correctly. e.g. `(()(()))` is nested correctly, but `(()(` is not.

   Use a stack. Push something on **if** you find (, pop something off **if** you find). Return **true if** stack is empty at the end.

2. Given a text file that contains the name and country of birth of each Berkeley student, count the number of different countries that are represented.

   Use a set. Add every birth country in the text file to the set. Since the set does not add duplicates, the total number of items at the end will be the number of different countries.

3. Given a bunch of strings, partition the strings into groups based on which ones are anagrams of each other. e.g. given {"cat", "love", "act", "bat", "tab", "tac"}, you should return {{"cat", "act", "tac"}, {"love"}, and {"bat", "tab"}}. Hint: assume you can write a helper method that takes in a string and returns the same string but with its characters sorted.

   Use a map. The keys into the map will be sorted strings, and the value associated with each one will the list of strings that are anagrams with it. So what you **do** is, **for** each string in the input, key into the map with its sorted version, and add the string to the list there.

3 More Complicated ADTs

The first page introduced you to some basic ADTs; you can find implementations of these in Java's standard library. But if we want something more complicated, we'll have to build it ourselves.

1. Suppose we want an ADT called `BiDividerMap` with the following functionality (assume `K` is something `Comparable`):

   ```
   put(K, V); // put a key, value pair
   getByKey(K); // get the value corresponding to a key
   getByValue(V); // get the key corresponding to a value
   numLessThan(K); // return number of keys in the map less than K
   ```

   Describe how you could implement this ADT building off the ADTs given on the first page. Do not write code. Then, suppose you decide you want `numLessThan(K)` to run really fast. Can you think of any ways to improve your implementation to account for this?

   Create two maps, one from K->V, and the other from V->K. Note that when you call put into the BiDividerMap, you have to then put into each of the two component maps.
   When numLessThan(K) is called, get the list of keys, sort it, and then iterate until you find a key that's bigger than the input

For the improvement:
Add another map from key to rank as well as a cached boolean. When put is called, set cached=false. When numLessThan() is called, if cached is true, use rankMap, otherwise rebuild rankMap and set cached=True.

2. Next, Suppose we would like to invent a new ADT called `MedianFinder` which supports the following operations:

```
add(int x); // add the integer into the collection
getMedian(); // returns the median integer in the collection
```

Again, describe how you could implement this ADT building off of the ADTs from the first page.

Use a list. When you add, just insert to the back of the list. When getMedian is called, first sort the list. Then figure out the size of the list and get the middle item.

Auxiliary for Adepts: Ensure that `add(int x)` and `getMedian()` each use a number of method calls independent of the items in the `MedianFinder` object.
Hint: Use two priority queues, one **for** the items less than the median and one **for** the items greater than the median.

4 ADTing in Circles

You want to solve a problem using a queue, but unfortunately, you only have access to a class that is a stack. You decide to implement the queue ADT just using stacks. Complete the following class, assuming that you have access to a class called `Stack` which implements the stack ADT. Hint: Consider using two stacks.

```
public class SQueue {
    private Stack inStack;

    public SQueue() {
        inStack = new Stack();
    }

    public void enqueue(int item) {
        Stack tempStack = new Stack();
        while (!inStack.isEmpty()) {
            tempStack.push(inStack.pop());
        }
        inStack.push(item);
        while (!tempStack.isEmpty()) {
            inStack.push(tempStack.pop());
        }
    }

    public int dequeue() {
        return inStack.pop();
    }
}
```

Auxiliary for Adepts: Can you do it with only one stack? **Especially Extra:** Are you really getting away with using only one stack?

```
You can do it with one stack and recursion. And no, because recursion uses a
   stack of method calls!

public class SQueue {
    private Stack inStack;

    public SQueue() {
        inStack = new Stack();
    }

    public void enqueue(int item) {
        inStack.push(item);
    }

    public int dequeue() {
        return dequeueHelper(inStack.pop());
    }

    // Recursive helper method
    private int dequeueHelper(int prev) {
        if (inStack.empty()) {
            return prev;
```

```
        }
        int cur = inStack.pop();
        int temp = dequeueHelper(cur);
        enqueue(prev);
        return temp;
    }
}
```