Rectangles: Version I

1. Add methods to SimpleRectangle that return the area and the perimeter of the rectangle (add two methods, one for each property).

2. Add a top-left point to SimpleRectangle so that rectangles are anchored in the plane (assume integer coordinates). Add a new constructor that accepts four parameters and provide default values in the original two-parameter constructor.

3. Add a point to a rectangle so that it grows minimally to contain the new point. Don't use if statements, use static methods Math.min and Math.max to make calculations.

```java
public add(int x, int y)
{
}
```

Develop some test cases with anticipated results to see if your method works correctly.

4. Write an equals method to test if one rectangle is equal to another:

```java
public boolean equals(Object o)
{
    SimpleRectangle = (SimpleRectangle) o;
    // add code here

}
```

5. Write a method that finds the perimeter of a java.awt.Rectangle object (assume your method is a static method in a class named RMath). You'll need to consult the API provided.

```java
public static int perimeter(Rectangle r)
{
}
```

6. Given three java.awt.Rectangle, return the rectangle that is their intersection, use the intersection method rather than createIntersection.

```java
public static Rectangle intersection(Rectangle r1, Rectangle r2, Rectangle r3)
{
}
```

7. Do the same problem as above, but assume an array of Rectangles is passed to the function (return intersection of all the rectangles).
public static Rectangle intersection(Rectangle[] list)
{
}

BlueJ

8. Using BlueJ and BigFactorial.java determine how many digits there are in 76! (call a method that returns the result without writing new code).

Priority Queues

These questions make reference to the AP PriorityQueue interface and the ArrayList-based simple implementation in ArrayPriorityQueue.java.

ArrayPriorityQueue

9. The code below show methods peekMin and removeMin from the class ArrayPriorityQueue.

What is the big-Oh complexity of for an N-element priority queue of removeMin and of peekMin? Justify.

/**
 * Removes and returns a minimal element of this pq
 * @return a minimal element after removing it
 */
public Object removeMin()
{
    Object min = peekMin();
    items.remove(min);
    return min;
}

/**
 * Returns a minimal element of this pq
 * @return a minimal element
 */
public Object peekMin()
{
    int minIndex = 0;
    for (int i = 1; i < items.size(); i++) {
        Comparable c = (Comparable) items.get(i);
        if (c.compareTo(items.get(minIndex)) < 0) {
            minIndex = i;
        }
    }
    return items.get(minIndex);
}

10. What happens if either one of peekMin or removeMin is called on an empty priority queue?
11. Consider this alternative implementation of removeMin. Briefly give a reason why this implementation is superior to the one shown above and a reason that it is inferior.

```java
/**
 * Removes and returns a minimal element of this pq
 * @return a minimal element after removing it
 */
public Object removeMin()
{
    int minIndex = 0;
    for(int k=1; k < items.size(); k++) {
         Comparable c = (Comparable) items.get(k);
        if (c.compareTo(items.get(minIndex)) < 0) {
            minIndex = k;
        }
    }
    Object retval = items.get(minIndex);
    items.set(minIndex,items.get(items.size()-1));
    items.remove(items.size()-1);
    return retval;
}
```

12. Consider the method sort below that sorts its parameter into non-decreasing order. What is the big-Oh complexity of this method? How does the complexity change if the formal parameter list has type LinkedList (in which case the function still works correctly)?

```java
/**
 * Sorts list into ascending/non-decreasing order.
 */
public void sort(ArrayList list)
{
    PriorityQueue pq = new ArrayPriorityQueue();
    int size = list.size();
    for(int k=0; k < size; k++) {
        pq.add(list.get(k));
    }
    for(int k=0; k < size; k++) {
        list.set(k, pq.removeMin());
    }
}
```

---

Owen L. Astrachan
Last modified: Sat Mar 29 02:28:17 EST 2003