Exer 1, Lecture 11: There are a number of programming practices that are known to lead to overly complex programs that are difficult to understand and debug. One of these is the use of “goto” statements; another is the use of “break” statements in the body of a loop to terminate the loop. Programs are generally easier to understand and to reason about if and only if loops are terminated when the loop condition is false.

Redo the `insertItem()` for a heap data structure that is given on p. 130 of the text so that the loop is programmed as described above without a `break` statement and without using any new boolean variables.

Answer:

```java
public void insertItem(Object k, Object e) throws InvalidKeyException{
    if (!comp.isComparable(k))
        throw new InvalidKeyException("Invalid Key");

    Item newItem = new Item(k,e);
    Position u, z = T.add(newItem);
    // Move items down the tree until find place for newItem
    while( !T.isRoot(z) && comp.isLessThan(k,key(T.parent(z))) ){
        u = T.parent(z);
        T.replaceElement(z,u.element());
        z = u;
    }

    // Set u to position for newItem and put newItem there
    T.replaceElement(z, newItem);
}
```

Comment on above code: In the loop above, the method `T.parent(z)` is called twice each time the body of the loop is executed. This can be eliminated by moving the assignment statement for `u` into the loop condition.
as shown below.

```java
while( !T.isRoot(z) && comp.isLessThan(k,key(u=T.parent(z))) ){
    T.replaceElement(z,u.element());
    z = u;
}
```

The choice between these two loop forms is a matter of personal taste. Pedagogically, I prefer the former; in actual practice, I might use the latter depending on how crucial speed is in the application.

While we are eliminating break statements, we should show also how to eliminate the break statement in the method `removeMin()` on p. 130.

When we insert a new item in a heap with the program above, we put it “at the bottom” and “bubble it up.” We stop when we either get to the top (i.e., root) of the heap or we find the correct place to insert the new node in the “middle” of the heap. We find this by comparing keys and by moving up the heap using the `parent()` method.

When we do the `removeMin` operation, as in the program below, we take the minimum element from the root and then swap an item from the bottom of the heap to the root and then “bubble it down.” We stop when we reach the bottom of heap (as opposed to the top) or we find the correct place to put it in the “middle” of the heap. We find this by comparing keys and by moving down the heap.

The method `isRoot()` tells us when we are at the top of a heap. For `removeMin()`, we need an analogous method that tells us when a path has reached the bottom of the heap, so we introduce one that we call `isHeapBottom()`. It is a member of the class `HeapPriorityQueue` (see p. 129 of G & T). Here is the code

```java
boolean isHeapBottom(Position p){
    return isExternal(T.leftChild(p));
}
```

In the `insertItem()` method, we use the `parent()` method to work our way up the binary tree representation of the heap. In the `removeMin()` method, we need to work our way down the tree and we need a method analogous to `parent()`. We add to the class `HeapPriorityQueue` another method named `minChild()` with the signature

```java
public Position minChild(Position p)
```

where `p` is the position of a node in the `HeapTree` that has a least one internal child, that is, it is not at the bottom of the heap. It returns the position of the child of `p` that has the smallest key.
public Position minChild(Position p) {
    Position minChld = T.leftChild(p),
    rChild = T.rightChild(p);
    if ( T.isInternal(rChild) &&
        comp.isLessThan(key(rChild), key(minChld)) ){
        minChld = rChild;
    }
    return minChld;
}

Now with these two methods the code for the new version of removeMin() is very analogous to the code for insertItem().

public Object removeMin() throws PriorityQueueEmptyException{
    if (isEmpty())
        throw new PriorityQueueEmptyException("Empty Priority Queue");

    Object min = (T.root()).element();
    Item item = T.remove();
    Object k = key(item);
    Position z = T.root(),
    if (size() > 1){
        Position u;
        while(!T.isHeapBottom(z) && // Find place to insert item
                comp.isLessThan(key(T.minChild(z)), k) ){
            u = T.minChild(z);
            T.replaceElement(z,u.element());
            z = u;
        }
        T.replaceElement(z,item);

        return min;
    }

    The two calls of T.minChild(z) can be replaced by a single call in a way analogous to the way that a call of T.parent(z) was eliminated in the code for insertItem() above.

    B. F. Caviness
    Univ. of Delaware