Answers to Self-Study Questions

Test Yourself #1

For each of the three proposed data structures (an array, a linked list, and a BST) we must describe the general approach and we must say how each of the operations (create an empty priority queue, insert a priority, return the highest priority, remove the highest priority, and say whether the priority queue is empty) will be implemented.

Using an Array

If we use an array to store the values in the priority queue, we can either store the values in sorted order (which will make the insert operation slow and the removeMax operation fast) or in arbitrary order (which will make the insert operation fast and the removeMax operation slow). Note also that we'll need a field to keep track of the number of values currently in the priority queue, we'll need to decide how big to make the array initially, and we'll need to expand the array if it gets full. Here's a partial class definition:

```
public class PriorityQueue {
    // *** fields ***
    private Comparable[] queue;
    private int numItems;
    private static final int INIT_SIZE = 10;

    // *** constructor ***
    public PriorityQueue() {
        queue = new Comparable[INIT_SIZE];
        numItems = 0;
    }

    ...
}
```

As mentioned above, if we keep the array sorted, then the insert operation is worst-case O(N) when there are N items in the priority queue: we can find the place for the new value efficiently using binary search, but then we'll need to move all the larger values over to the right to make room for the new value. As long as we keep the array sorted from low to high, the removeMax is O(1) -- just decrement `numItems` and return the last value. (Similarly, getMax is O(1) -- just return the item at position `numItems - 1`.)

If we keep the array unsorted, then insert is O(1) (ignoring the time to expand the array) but removeMax is O(N) since we must search the whole array for the largest value. For a similar reason, getMax is O(N) as well.

The isEmpty operation is trivial -- just return true iff `numItems == 0`, so it (and the constructor) are both O(1).

Using a Linked List

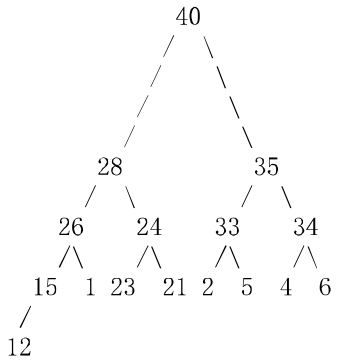
Using a linked list is similar to using an array; again, we can keep the list sorted (which makes insert O(N)) or unsorted (which makes removeMax O(N)). Note that if the list is sorted we must use linear search to find the place to insert the new value but there is no need to move old values over. Also, unless we maintain a "tail" pointer, we should keep the list in order from high to low, since removing from the head of the list can be done in O(1) time (and getMax is also O(1) -- just return the value at the head of the list). Note that for a linked list we don't need the `numItems` field; the isEmpty operation returns true iff the list is empty (which can be determined in O(1) time).

Using a BST

If we use a BST, then the largest value can be found in time proportional to the height of the tree by going right until we reach a node with no right child. Removing that node is also O(tree height) as is inserting a

new value. If the tree stays balanced, then these operations are no worse than O(log N); however, if the tree is not balanced, insert, getMax, and removeMax can be O(N). As with the linked-list implementation, there is no need for a `numItems` field.

Test Yourself #2



Test Yourself #3