Data structures DAT037 – exam answers
15th January 2019

Note: I've added explanations to the answers here even when they're not required by the question.

Question 1

a) **For a 3:** $O(n^2)$. Each loop runs at most n times, and in the worst case (when S becomes unbalanced) each BST operation takes $O(n)$ time (because S contains at most n elements). **For a 4/5:** $O(m(m+n))$. The first loop runs m times, the second runs n times, and each BST operation takes $O(m)$ time in the worst case. Alternatively, $O(m \max(m,n))$.

b) **For a 3:** $O(n \log n)$. Similar to part a, but each BST operation takes $O(\log n)$ time. **For a 4/5:** $O((m+n) \log m)$. Similar to part a, but each BST operation takes $O(\log m)$ time. Alternatively, $O(\max(m,n) \log m)$.

For a 5: looking at the complexity above, you should make sure that $m \leq n$. You can do this by calling `isDisjoint(X,Y)` if X is smaller than Y, and `isDisjoint(Y,X)` if Y is smaller than X. **Note:** it wasn't clear in the question if you should answer this part assuming a binary search tree or an AVL tree. Either is acceptable (in any case, both give the same answer).

Question 2

a) 9, 11 or 10. For all other values, there is a blank space coming before index 1 which would have been used.

b) When searching for a value, if a blank space is encountered then the search returns `false` (the value is not found). If an **XXX** is encountered then the search continues onto the next array index.

c) Take an empty hash table of size 10, and insert 10 and then 20 into it. 10 will be stored at index 0, and 20 will be stored at index 1. Now delete 10, so that there is a blank space at index 0. A search for 20 will then fail (index 1 will not be searched).

Question 3

```
public void setParents() {
  setParentsFrom(root, null);
}

// A helper method that takes the node and its parent as parameters.
private void setParentsFrom(Node node, Node parent) {
  if (node == null) return;
  node.parent = parent;
  setParentsFrom(node.left, node);
  setParentsFrom(node.right, node);
}
```

The algorithm takes linear time because it visits each node exactly once and does a constant amount of work when visiting each node.

Question 4

For a 3

You can solve this using a binary heap. `new()` creates an empty heap. `add(x)` inserts x into the heap using ordinary heap insertion.

The idea with `findSecondSmallest()` and `deleteSecondSmallest()` is to first remove the minimum element from the heap, so that the second-smallest element will be the new minimum element. We can read or remove that element and then re-add the previous minimum element.

`findSecondSmallest()` can be implemented like so, assuming that `heap` is the binary heap:

```
x = heap.findMin(); // find and remove the smallest element
heap.deleteMin();
y = heap.findMin(); // y was originally the second-smallest element
heap.add(x);        // re-add the smallest element
return y;
```

`deleteSecondSmallest()` can be implemented like so:

```
x = heap.findMin();
heap.deleteMin();
heap.deleteMin();
heap.add(x);
```

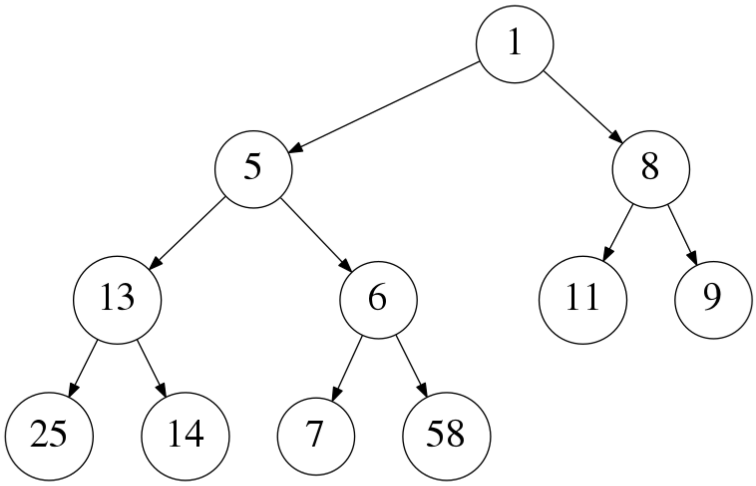
For a 4 or 5

We maintain three pieces of data: a Boolean `empty` that is true if the priority queue is empty, an integer `min` that holds the smallest element in the priority queue (its value is only used if `empty` is false), and a binary heap `heap` that holds all the other elements.

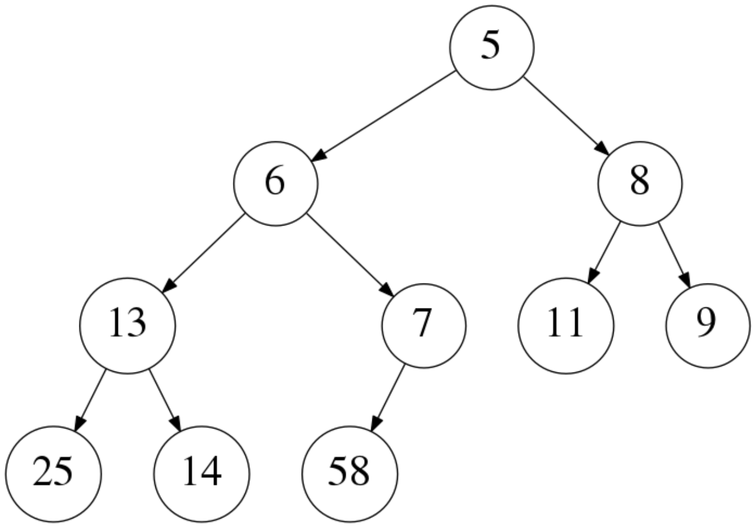
```
new() {
  empty = true;
  heap = new empty binary heap;
}

add(x) {
  if (empty) {
    empty = false;
    min = x;
  } else {
    if (x < min) {
      heap.insert(min);
      min = x;
    } else {
      heap.insert(x);
    }
  }
}

findSecondSmallest() {
  return heap.findMin();
}

deleteSecondSmallest() {
  heap.deleteMin();
}
```

Question 5

a) Only C is a binary heap. (In A, 7 > 5 violates the heap invariant; in B, 21 > 20 violates the heap invariant.)

b) 

c) You end up with the following tree:



As an array, the answer is: 5 6 8 13 7 11 9 25 14 58. (Writing the answer either as a tree or an array is acceptable.)

Question 6

For a 3

Here is a possible solution, which sorts the array and then remembers which values have already been printed:

```
sort x
S = new empty set (implemented using e.g. an AVL tree)
for each n in x
  if not S.member(n)
    print(n)
    S.insert(n)
```

Here is another one, which exploits the fact that duplicate values will be next to each other in the array after sorting:

```
sort x
print(x[0])
for i from 1 to x.length-1
  if x[i-1] != x[i]
    print(x[i])
```

Here is another variation, which remembers the value that was most recently printed:

```
sort x
lastPrinted = null
for each n in x
  if n != lastPrinted
    print(n)
    lastPrinted = n
```

For a 4 or 5

Here is one approach, which uses the fact that inorder traversal of a binary tree gives the values out in ascending order:

```
S = new AVL tree
for each n in x
  S.insert(n)
visit the nodes of S using an inorder traversal and print each one
```

Here is another, which first eliminates duplicates and then sorts:

```
S = new hash table
for each n in x
  S.insert(n)
copy contents of S into a new array, y
sort y
for each n in y
  print(y)
```