Compsci 100

Owen Astrachan

April 14, 2010

Name: _____

Login: _____

Honor code acknowledgment (signature) _____

	value	grade
Problem 1	10 pts.	
Problem 2	24 pts.	
Problem 3	22 pts.	
Problem 4	16 pts.	
Problem 5	16 pts.	
TOTAL:	88 pts.	

This test has 12 numbered page (there's a three-page insert at the end that isn't numbered), be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.

In writing code you do not need to worry about specifying the proper `import statements`. Assume that all libraries and packages we've discussed are imported in any code you write.

Unless indicated otherwise, the `TreeNode` class for this test is on the left. Some common recurrences and their solutions are on the right.

```
public static class TreeNode {
    String info;
    TreeNode  left;
    TreeNode  right;
    TreeNode(String val,
             TreeNode  lptr,
             TreeNode  rptr) {
        info = val;
        left = lptr;
        right = rptr;
    }
}
```

label	**recurrence**	**solution**
A	`T(n) = T(n/2) + O(1)`	$O(\log n)$
B	`T(n) = T(n/2) + O(n)`	$O(n)$
C	`T(n) = 2T(n/2) + O(1)`	$O(n)$
D	`T(n) = 2T(n/2) + O(n)`	$O(n \log n)$
E	`T(n) = T(n-1) + O(1)`	$O(n)$
F	`T(n) = T(n-1) + O(n)`	$O(n^2)$
G	`T(n) = 2T(n-1) + O(1)`	$O(2^n)$

PROBLEM 1 : (*Easy Does It (10 points)*)

Part A (4 points)

The values $1, 2, \ldots, n$ are added to an initially empty queue in that order, then $n/2$ values are removed from the queue using the queue remove operation. What is the big-O runtime of this sequence of queue operations where queues are implemented using doubly-linked lists with pointers to the first and last nodes; justify your answer briefly.

Part B (4 points)

Suppose the same sequence of queue operations are performed, but queues are implemented with singly-linked lists and pointers to the first and last nodes. What is the big-O runtime; justify your answer.

Part C (2 points)

Give the value of the postfix expression `4 5 6 + *`

PROBLEM 2 : (***Amazing Max (24 points)***)

Part A (4 points)

The method `max` below correctly returns the largest `int` value in an array. What is the big-O runtime complexity in terms of n where n is the number of elements in the array. Justify your answer with a recurrence.

```
public int max(int[] a){
    return rmax(a,0);
}

public int rmax(int[] a, int firstIndex){
    if (firstIndex == a.length - 1){
        return a[firstIndex];
    }
    return Math.max(a[firstIndex], rmax(a,firstIndex+1));
}
```

Part B (4 points)

The method `stmax` below correctly returns the largest element in a *binary search tree*. What is its runtime complexity in both the average and the worst case. Justify your answer. *Be sure to provide two answers and two justifications, do not use recurrences: the method isn't recursive.*

```
public String stmax(TreeNode root){
    while (root.right != null){
        root = root.right;
    }
    return root.info;
}
```

Part C (4 points)

The method below correctly returns the largest element in a min-heap, where the heap is stored in an array as we discussed in class: the children of the element at index k are at indexes $2k$ and $2k+1$. Explain two things: (1) why the method works and (2) what its complexity is for an n element heap. The root of the heap is stored at index 1. Assume the array is full, i.e., every value stored in the array is a value in the heap (except for the value at index 0).

```
public int findMax(int[] heap) {
    int lastNonLeafIndex = (heap.length-1)/2;
    int max = heap[lastNonLeafIndex];
    for(int k=lastNonLeafIndex+1; k < heap.length; k++){
        max = Math.max(max,heap[k]);
    }
    return max;
}
```

Part D (6 points)

The method `trmax` below returns the maximal value stored in a binary tree, regardless of whether the tree is a search tree. What is the big-O complexity of `trmax` in **both** the average and worst case for an n-node tree. Justify your answer with recurrences as appropriate, other justifications are fine for this problem.

```
public String max(String a, String b) {
  if (a.compareTo(b) <= 0) return b;
  return a;
}
public String trmax(TreeNode root){
  if (root == null) return "";
  String self = root.info;
  return max(max(self, trmax(root.left)),trmax(root.right));
}
```

Part E (4 points)

Method `pmax` below returns the largest value in an array by using a priority queue. What is the big-O complexity of `pmax` for an n element array in the **worst case**? Justify your answer.

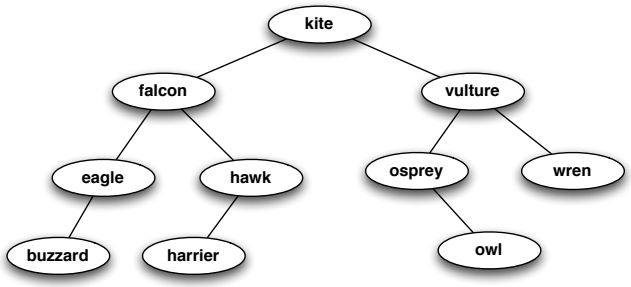
```
public String pmax(String[] list){
  PriorityQueue<String> pq = new PriorityQueue<String>(list.length,Collections.reverseOrder());
  pq.addAll(Arrays.asList(list));
  return pq.remove();
}
```

Part F (2 points)

Explain in words or code how to modify `pmax` above to return the k-th largest element, where k will be a parameter to `pmax` and $k = 0$ means the largest element while $k = list.length - 1$ means the smallest element.

PROBLEM 3 : (*One Two, Tree (22 points)*)

This problem makes reference to the **search tree** below.



Part A (2 points)

In a preorder traversal of the tree above the last value visited/printed is *wren*. What is the second value visited/printed in a preorder traversal?

Part B (3 points)

Write the first three values of a *postorder* traversal of the tree.

Part C (6 points)

Add the values *kestrel*, *condor*, and *ostrich* (in that order) to the tree above so that it remains a search tree. Draw the new values linked appropriately.

Part D (6 points)

In this problem don't include the values you added in **Part C**.

Suppose *hawk* is the root of the entire tree instead of *kite* (as shown) but the tree is still a search tree, i.e., the elements are rearranged so that there is a new root: *hawk*. How many different ways are there to structure the left subtree of *hawk* so that it remains a search tree? Justify your answer. For example, only the values *falcon*, *harrier*, *eagle*, and *buzzard* could be the root of the left subtree of *hawk*.

How many ways are there to structure the right subtree of *hawk*? Justify your answer.

How many different binary search trees are there with *hawk* as the root? Justify your answer.

The first sixteen Catalan numbers are listed below.

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845.

(You should have three answers for this problem.)

Part E (5 points)

Write methods `printLevel` and `levelCount` that print and count the nodes at a specified level where the global root is at level zero/0. For printing, the nodes should be printed from left-to-right at a given level.

For example `printLevel(root,3)` should result in printing the following values, one per line, in the order shown, assuming `root` points to the *kite* node in the tree diagrammed above.

buzzard harrier owl

Complete `printLevel` below (hint: when level is zero, print).

```
public void printLevel(TreeNode root, int level) {









}
```

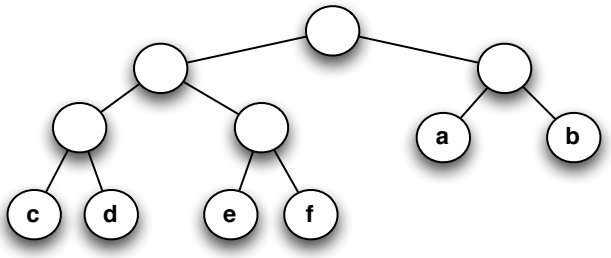
Complete `levelCount` below.

The call `levelCount(root,2)` should return 4, there are four nodes at level 2: *eagle*, *hawk*, *osprey*, and *wren*.

```
public int levelCount(TreeNode root, int level) {









}
```

PROBLEM 4 : (*Big Bad Wolf (16 points)*)

You'll use the Huffman tree below in several parts of this problem.



Part A (2 points)

For this tree, with 0 indicating left and 1 indicating right, what is the Huffman code for the letter/chunk 'e'?

Part B (4 points)

Decode the string encoded by the bit sequence: 1 1 1 0 0 0 1 0 1 1 0 1 0 0 1 0 0 0 1 (show reasoning for partial credit)

Part C (4 points)

In Huffman coding a map is created that maps each character/chunk to its encoding. Values are added to the map by traversing the Huffman tree and storing every root-to-leaf path as the value in the map for the key stored in the leaf. The method `loadMap` below does this, it would be called as `loadMap(myRoot,"")` where `myRoot` is the root of the Huffman tree. The non-recursive part of `loadMap` has been completed. Add two lines to complete the method, both should be recursive calls. In the code below the non-recursive case stores values in an instance variable `myMap`, but the recursive calls don't reference any instance variables. You can assume every internal node has exactly two children in any Huffman tree.

```
public void loadMap(TreeNode root, String path) {

    if (root.left == null && root.right == null) {
        myMap.put(root.myValue, path);
    }
    else {









    }
}
```

Part D (6 points)

The method `map2tree` below uses a helper method you'll complete. The method recreates the Huffman tree from the encodings stored in a map such as the one created by method `loadMap` in the previous problem. The helper method is called once for every key/value in the map: the key from the map is the chunk stored in the leaf of a Huffman tree whose root-to-leaf path is the value from the map. You'll complete method `helper`. The code is very similar to the `TrieLexicon` code from the Boggle assignment since the Huffman tree is actually a trie with 0/1 branching. The code to add a string to the `TrieLexicon` is at the end of this test as is the code for the class `TreeNode` from the Huffman assignment.

```
public TreeNode map2tree(){
    TreeNode root = new TreeNode(0,0,null,null);
    for(int chunk : myEncodings.keySet()) {
        root = helper(root, chunk, myEncodings.get(chunk));
    }
    return root;
}
/**
  * Add chunk to a leaf as specified by path in the Huffman tree
  * whose root is root.
  * @param root is the root of the Huffman tree
  * @param chunk is the value stored in the leaf
  * @param path is the root-to-leaf path
  * @return the root of the Huffman tree
  */
private TreeNode helper(TreeNode root,int chunk, String path){

    TreeNode current = root;
    for(int k=0; k < path.length(); k++){
        // you fill in code here









        }
    }
    current.myValue = chunk;
    return root;
}
```

PROBLEM 5 : (***BST redux (16 points)***)

For this problem binary trees store integer values instead of strings and no duplicate values appear in the tree.

```
public class IntNode{
  int info;
  IntNode left,right;
  public IntNode(int value){
       info = value;
  }
}
```

By definition a *binary search tree* is a binary tree if each of the following properties hold for every node in the tree.

1. All values in the left subtree of a node are less than the node's value.
2. All values in the right subtree of a node are greater than the node's value
3. Both the left and right subtree of a node are binary search trees.

Part A (2 points)

Draw a binary tree whose root satisfies the first two properties, but not the third, so that it's not a search tree. Nodes contain integer values.

Part B (2 points)

Draw a binary tree whose root satisfies the second two properties, but not the first, so that it's not a search tree. Nodes contain integer values.

Part C (6 points)

The methods `getMin` and `getMax` below return the smallest and largest values in a tree, respectively. They are used in writing `isSearchTree` that returns a boolean value indicating whether a tree is a search tree using the definition of a search tree given above. In reasoning about `isSearchTree` assume that both `getMin` and `getMax` are $O(n)$ in both the average and worst case, *you don't need to write recurrences for* `getMin` *and* `getMax`.

```
public int getMax(IntNode root){
    if (root == null) return Integer.MIN_VALUE;
    return Math.max(root.info,
           Math.max(getMax(root.left), getMax(root.right)));
}
public int getMin(IntNode root){
    if (root == null) return Integer.MAX_VALUE;
    return Math.min(root.info,
           Math.min(getMin(root.left), getMin(root.right)));
}

public boolean isSearchTree(IntNode root){
    if (root == null) return true;
    return
      isSearchTree(root.left)  &&
      isSearchTree(root.right) &&
        root.info > getMax(root.left) &&
        root.info < getMin(root.right);
}
```

What is the runtime of `isSearchTree` in both the average and the worst case for a tree of n nodes? You must provide two recurrence relations: one for the average case in which trees are balanced, and one in the worst case in which trees are completely unbalanced.

(continued with **Part D** on next page)

Part D (2 points)

A student suggests different implementations of `getMax` and `getMin` because in their use in the code above they are only called when it is known that the tree is a search tree. The alternate implementations are below.

```
public int getMax(TreeNode root){
    if (root == null) return Integer.MIN_VALUE; // special case
    while (root.right != null){
        root = root.right;
    }
    return root.info;
}
public int getMin(TreeNode root){
    if (root == null) return Integer.MAX_VALUE; // special case
    while (root.left != null){
        root = root.left;
    }
    return root.info;
}
```

Explain what the recurrences would be in the average and worst case for `isSearchTree` using these alternative implementations of `getMin` and `getMax`. You do not need to solve the recurrences, you just need to give them.

Part E (4 points)

Another student says that the alternative implementations can be used even when the body of `isSearchTree` is below, where the recursive calls are swapped with the calls to `getMin` and `getMin`. In a few words indicate whether you think this code will work correctly with the alternative implementations and why you think this.

```
public boolean isSearchTree(IntNode root){
    if (root == null) return true;
    return
        root.info > getMax(root.left)  &&
        root.info < getMin(root.right) &&
        isSearchTree(root.left)  &&
        isSearchTree(root.right);
}
```

(nothing on this page)

```
public class TreeNode implements Comparable<TreeNode> {

    public int myValue;
    public int myWeight;
    public TreeNode myLeft;
    public TreeNode myRight;

    public TreeNode(int value, int weight) {
        myValue = value;
        myWeight = weight;
    }

    public TreeNode(int value, int weight, TreeNode ltree, TreeNode rtree) {
        this(value, weight);
        myLeft = ltree;
        myRight = rtree;
    }

// more code here for TreeNode, not shown
}


public class TrieLexicon implements ILexicon {

    public static class Node {
        String info;
        boolean isWord;

        Map<Character,Node> children;
        Node parent;
        Node(char ch, Node p) {
            info = ""+ch;
            isWord = false;
            children = new TreeMap<Character,Node>();
            parent = p;
        }
    }

    protected Node myRoot; // root of entire trie

    public TrieLexicon() {
        myRoot = new Node('x', null);
        mySize = 0;
    }


    public boolean add(String s) {
        Node t = myRoot;

        for (int k = 0; k < s.length(); k++) {

            char ch = s.charAt(k);
            Node child = t.children.get(ch);
            if (child == null) {
               child = new Node(ch, t);
               t.children.put(ch,child);
            }
            t = child;
        }

        if (!t.isWord) {
            t.isWord = true; // walked down path, mark this as a word
            mySize++;
            return true;
        }
        return false; // already in TrieLexicon
    }
        // more code here not shown
}
```