CS2223 Algorithms **B Term 2015**

Exam 2 **December 17, 2015**

NAME: ___SOLUTIONS________

Instructions:

- Time allowed: 50 minutes
- Show your work and justify your answers
- Use the space provided to write your answers

Total	Q1	Q2	Q3	Q4	Q5

DO NOT OPEN EXAM UNTIL INSTRUCTED TO DO SO!!

Question 1. (20 pts.) Short Answer Questions

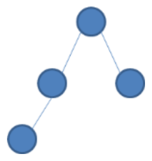
For each of the following statements: TWO POINTS for correct. THREE points for explanation

- If the statement is *true*, circle **True** and explain why.
- If the statement is *false*, circle **False** and explain *why the statement is false.*

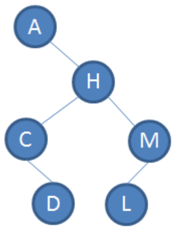
Your explanations should be ***brief*** (using about *one sentence*), but complete.

(a) [5 pts.] **True / False**: Given an AVL binary search tree, the number of nodes in the root's left subtree is the same as the number of nodes in the root's right subtree.

FALSE: AVL only guarantees heights of left and right children are within +/- of each other. Sample tree is balanced but has different # of nodes in left and right:



(b) [5 pts.] Construct a binary search tree that produces the following output in this exact sequence when conducting a preorder traversal of the tree: **A H C D M L**

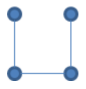


(c) [5 pts.] **True / False:** Given a heap-based priority queue, **pq**, of size N, a **contains(x)** method that checks whether **x** is contained by **pq** requires O(N log N).

FALSE: The heap is stored in an array. You can search an array for a given element in time O(N) where N is the number of elements in the array.

(d) [5 pts.] **True / False:** N is an even number greater than 2. It is impossible to construct a **connected** undirected graph of N vertices where N/2 vertices have degree 1.

FALSE: You can construct a graph as follows. Half of nodes at top with one edge done. Other half of nodes in long connected line.



Question 2. (20 pts.) Timing Analysis

Assume an AVL **balanced binary search tree** structure of `int` keys with the following methods:

```
public class AVL {
  Node root;

  class Node {
    int    key;
    Node   left, right;
    int    height;      // height from most distant-leaf in subtree
  }

  /* In subtree rooted at n, return count of values within range [left, right]. */
  int within(Node n, int left, int right) {
    if (n == null) { return 0; }

    int ct = within(n.left, left, right) + within(n.right, left, right);
    if (left <= n.key && n.key <= right) { ct++; }
    return ct;
  }

  /* For k > 1. */
  public int special (int k) {                        frequency       Cost
    int ct = 0;                                          _1_           _1_
    while (k > 1) {                                    _log k_         _1_
      ct += within (root, k/2, (3*k)/2);               _log k_         _N_
      k /= 2;                                          _log k_         _1_
    }

    ct += within (root, ct-1, ct+1);                     _1_           _N_
    return ct;
  }
}
```

(a) **[10 pts.]** The **`special(k)`** method is executed on a balanced AVL tree with N nodes. For each of the 5 annotated statements above, determine the frequency of the statement and its cost (i.e., execution time).

Cost of the within() recursive method is N because it visits all N nodes, regardless of the values of left and right. ONE POINTS for each correct answer (there are ten of them).

(b) **[5 pts.]** Compute the aggregate cost when invoking **`special(k)`** on a balanced AVL tree with N nodes
Missing any of these components loses one point

Total aggregate is: 1 + 2*log k + N*Log k + N

(c) **[5 pts.]** What is the order of growth using Big O notation when invoking **`special(k)?`**

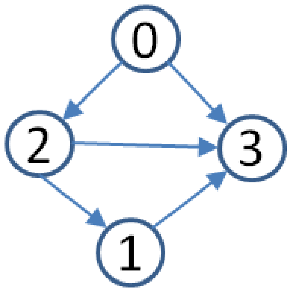
O(N log k) since N log k is always greater than N for k>1

Regardless of whether (b) is correct, your O(...) must match part (b).

Question 3. (20 pts.) Type Question

A directed graph class, **Digraph**, uses an adjacency list structure shown on the right. Add to **Digraph** a Java method **into(v)** that returns the vertices (should any exist) from which an edge starts and completes at **v**. The **into(v)** method must produce the following output for the sample graph below:

```
public class Digraph {
    final int V;
    int E;
    Bag<Integer>[] adj;
```



Sample Code	Sample Output

`// assume you have Digraph dg on left`
`for (int w : dg.into(3)) {`
`  StdOut.println(w);`
`}`
	0
	1
	2

(a) **[15 pts.]** You can provide Java code or clear pseudocode. Don't simply describe your answer in words.

Key idea is to go through all vertices that might have an edge into t and add to bag

```
public Iterable into (int t) {

   Bag<Integer> incoming = new Bag<Integer>();  +2 points
   for (int v = 0; v < V; v++) {  +2 points for loop over vertices
     if (v != t) { // not strictly necessary if digraph a simple graph
       for (int w : adj(v)) {      +3 points for loop over edges
          if (w == t) {          // found edge (v,w) that ends at t +3pt.
             incoming.add(v)     // be sure to add V to our bag
          }  +2 for adding to bag. +1 for being right vertex.
       }
     }
   }
   return incoming;  +2 for returning Bag.
}
```

b) **[5 pts.]** If a digraph has V vertices and E edges, write the order of growth using Big O notation for the runtime performance of **into(v)** that describes the worst case behavior.

This code visits every vertex and the adjacent edges to that vertex. Since this is a directed graph, this means that the innermost statements are executed E times. Now with a bag, we can assume constant time to add an element to a bag, thus the total time is O(V+E).

+5 for O(V+E). This is like DFS. +2 for O(V*E), or O(V) or O(E).

Question 4. (20 pts.) Algorithm Design

You are given a class **BST** on the right that implements a standard binary search tree. Add to this class a Java method **countLeaves()** that returns the total number of leaves contained in the tree. Recall that a leaf is a node in the tree with no left child and no right child.

Feel free to write additional methods as needed.

```
public class BST {
  Node root;

  class Node {
    int     key;
    Node    left, right;
  }
}
```

(a) **[12 pts.]** You can provide Java code or clear pseudocode. Don't simply describe your answer in words.

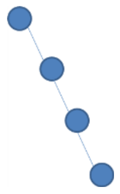
Key idea: write helper recursive method and check for leaves

```
public int countLeaves() {
     return countLeaves(root);    +2 helper method
}

int countLeaves(Node n) {     +2 helper method returns int
   if (n == null) { return 0; }   // NOT LEAF! Non-existent  +2 validate
   if (n.left == null && n.right == null) { return 1; }  // leaf! +2 points
   Return countLeaves(n.left) + countLeaves(n.right);  +2 pt (left) +2 pt (rt)
}
```

(b) **[4 pts.]** What is the fewest number of leaves in a BST that contains N keys? **Explain your answer**.

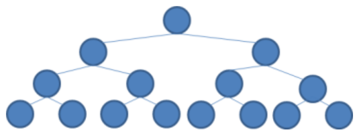
Consider a BST with N nodes that only has right-children. This has one leaf node.



+2 points for answer. +2 points for explanation

(b) **[4 pts.]** What is the most number of leaves in a BST that contains N keys? **Explain your answer**.

Consider a complete balanced tree, with every possible node in the fewest number of levels. In this



example below, there are 15 nodes and 8 leaves. If you look at a tree with seven nodes, there are four leaves. This suggests (N+1/2). So what if N is even? Note that you can add one more node here and still have the same number of leaves. In fact, you only increase the number of leaves by adding two nodes. So you could have Ceiling(N/2) or Floor(N+1/2).

+2 points for answer. +2 points for explanation.

Question 5. (20 pts.) Algorithm Question

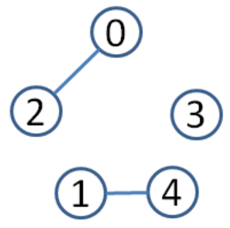
```
public class Graph {
  boolean marked[];
  final int V;
  int E;
  Bag<Integer>[] adj;

  public void dfsMain(int s) {
    for (int v = 0; v < V; v++)
      marked[v] = false;
    dfs (s);
  }

  void dfs (int v) {
    marked[v] = true;
    for (int w : adj[v])
      if (!marked[w]) { dfs(w); }
  }
```

(a) **[15 pts.]** Design a **connect()** method in the **Graph** class (shown on the right) that adds the fewest number of edges to a **disconnected** graph to make it connected **and** prints those edges to the console.

[To receive half credit on this part, you may add more edges than strictly necessary.]



```
Sample Code
// assume you have graph g
// on left
g.connect();
```

```
Sample Output
add edge (0,1)
add edge (0,3)
```

Key idea: DFS explores graph that can be reached. Simply add any new edge to unmarked and then you can repeat process from there. Make sure to call dfsMain first to initialize marked array.

```
public void connect() {
  dfsMain(0);                     // arbitrarily start at zero. +4 points
  for (int v = 1; v < V; v++) {   // for all other vertices…   +2 points
    if (!marked[v]) {              // +4 points find unmarked
      dfs(v);                     // explore onwards from here  +3 points
      print ("add edge(" + 0 + "," + v + ")");        +1 point
      addEdge(0,v);               // add edge back to assumed start +1 point
    }
  }
```

(b) **[5 pts.]** Classify the order of growth of the run-time performance of **connect()** using Big O notation. **Explain your answer.**

Note that DFS search requires O(V+E) to perform. You know this because it takes O(V) to simply mark initial vertex set, plus dfs(x) is called no more than once for each vertex. You must also account for edges E, which you can see from the inner for loop within dfs. Now look at the loop within connect. It will execute no more than V times. If the graph had initially been connected, dfsMain takes O(V+E) and the **for** loop would find all marked vertices in O(V) so the final time would still be O(V+E). But observe that if the graph is disconnected, then the successive dfs() calls within the for loop only explore unvisited vertices and edges because they are in fact disconnected from earlier marked vertices. By adding the edge **after** the call to dfs, we ensure we never traverse the same edge twice. Thus you can see that this revised code will never revisit the same vertex (because the for loop within connect starts at 1 and constantly increases) and it won't revisit the same edge, thus the overall performance is O(V+E). I will post sample solution code in the repository. +5 points if matches solution; explanations considered.

IF SOLUTION ADDS MORE THAN FEWEST # POSSIBLE, CAN GET 8 POINTS AS HALF-CREDIT