Name: Sample Solution

Email address: _____

CSE 373 Winter 2012: Midterm #1

(closed book, closed notes, NO calculators allowed)

Instructions: Read the directions for each question carefully before answering. We may give partial credit based on the work you **write down**, so if time permits, show your work! Use only the data structures and algorithms we have discussed in class or which were mentioned in the book so far.

Note: For questions where you are drawing pictures, please circle your final answer for any credit.

Good Luck!

Total: 72 points. Time: 50 minutes.

Question	Max Points	Score
1	16	
2	8	
3	19	
4	11	
5	6	
6	12	
Total	72	

1. (16 pts) **Big-O**

For each of the functions *f(N)* given below, indicate the tightest bound possible (in other words, giving $O(2^N)$ as the answer to every question is not likely to result in many points). Unless otherwise specified, all logs are base 2. **You MUST choose your answer from the following** (not given in any particular order), each of which could be re-used (could be the answer for more than one of a) – h)):

$O(N^2)$, $O(N^{1/2})$, $O(N^{1/4})$, $O(\log^3 N)$, $O(N)$, $O(N^2 \log N)$, $O(\log^8 N)$, $O(N^5)$, $O(2^N)$, $O(N^3)$, $O(N^8)$, $O(\log^4 N)$, $O(N \log^3 N)$, $O(\log^2 N)$, $O(\log N)$, $O(1)$, $O(N^3 \log N)$, $O(N^4)$, $O(N^N)$, $O(N^6)$, $O(N \log^2 N)$, $O(\log \log N)$, $O(\log^4 N)$, $O(N \log N)$

You do not need to explain your answer.

a) *f(N)* = $N^3 \cdot (4\log N - \log N) + (N^2)^2$ _____ $O(N^4)$_____

b) *f(N)* = $\log_8(2^{4N})$ _____ O(N) _____

c) *f(N)* = 500 N log N + N^2 log N _____ $O(N^2 \log N)$__

d) *f(N)* = (N· $(N + 1)^2$) ____ $O(N^3)$ ___

e) *f(N)* = log N^4 + $\log^2$ N ____$O(\log^2 N)$ _____

f) *f(N)* = N + log N + $N^{1/8}$ _____ O(N) _____

g) *f(N)* = $N^{1/4}$ + $(\log^4 N)^2$ _____ $O(N^{1/4})$ ____

h) *f(N)* = $N^3 \cdot N^2$ + $(N + 2N)^2$ _____ $O(N^5)$ _____

2. (8 pts) **Big-Oh and Run Time Analysis:** Describe the worst case running time of the following pseudocode functions in Big-Oh notation in terms of the variable `n`. ***Showing your work is not required*** (although showing work ***may*** allow some partial credit in the case your answer is wrong – don't spend a lot of time showing your work.). You MUST choose your answer from the following (not given in any particular order), each of which could be re-used (could be the answer for more than one of I. – IV.):

$O(n^2)$, $O(n^3 \log n)$, $O(n \log n)$, $O(n)$, $O(n^2 \log n)$, $O(n^5)$, $O(2^n)$, $O(n^3)$, $O(\log n)$, $O(1)$, $O(n^4)$, $O(n^n)$, $O(n^6)$, $O(n^8)$, $O(n^7)$

I.
```
int happy (int n, int m) {
    if (n < 10) return n;
    else if (n < 100)
      return happy (n - 2, m);
    else
      return happy (n/2, m);
}
```
Runtime: $O(\log n)$

II.
```
void sunny (int n) {
    j = 0;
    while (j < n) {
       for (int i = 0; i < n; ++i) {
           System.out.println("i = " + i);
           for (int k = 0; k < i; ++k)
              System.out.println("k = " + k);
       }
       j = j + 1;
    }
}
```
$O(n^3)$

III.
```
void smiley (int n) {
     for (int i = 0; i < n * n; ++i) {
        for (int k = 0; k < i; ++k)
            System.out.println("k = " + k);
        for (int j = n; j > 0; j--)
           System.out.println("j = " + j);
     }
}
```
$O(n^4)$

IV.
```
void funny (int n, int x) {
    for (int k = 0; k < 100; ++k)
       if (x > 500) {
          for (int i = 0; i < n * k; ++i)
             for (int j = 0; j < n; ++j)
                System.out.println("x = " + x);
       }
}
```
$O(n^2)$

3. (19 pts total) **Trees**.

a) (4 pts) What is the minimum and maximum number of **leaf** nodes in a **complete tree of height 6?** (Hint: the height of a tree consisting of a single node is 0) ***Give an exact number*** (with no exponents) for both of your answers – not a formula.

Minimum = $2^5 = 32$

Maximum = $2^6 = 64$

b) (4 pts) What is the minimum and maximum number of nodes in a balanced **AVL tree of height 6?** ***Give an exact number*** (with no exponents) for both of your answers – not a formula.

Minimum = 33

Maximum = $2^7 - 1 = 127$

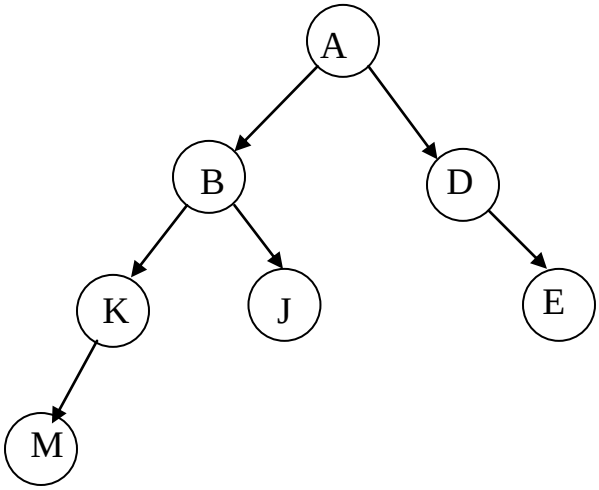
c) (2 pts) What is the maximum number of **leaf** nodes in a **full tree of height 5?** ***Give an exact number*** (with no exponents) for your answer – not a formula.

Maximum = $2^5 = 32$

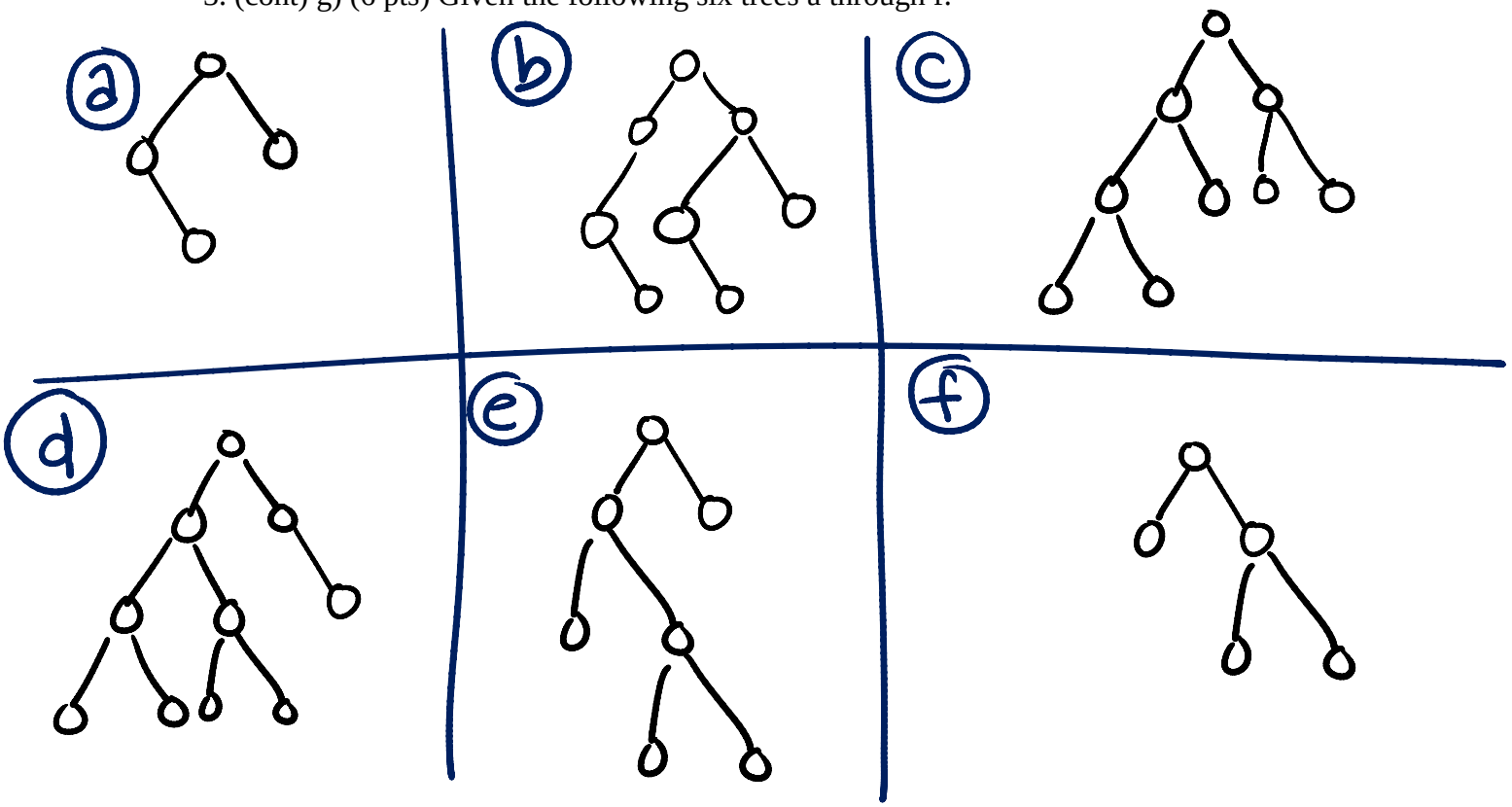
d) (1 pt) What is the **depth** of node K in the tree shown below: 2

e) (1 pt) Give a Pre-Order traversal of the tree shown below:
A B K M J D E

f) (1 pt) Is it AVL balanced (ignore the values, only look at the shape):
YES / NO



3. (cont) g) (6 pts) Given the following six trees a through f:



List the letters of all of the trees that have the following properties: (Note: It is possible that none of the trees above have the given property, it is also possible that some trees have more than one of the following properties.)

Complete: **_____c_____**

Full: **_____c, e, f_____**

AVL balanced: **_____a, c, d, f _____**

Perfect: **_____none_____**

4. (11 pts total) **Stacks – Please ANSWER PART (a) and PART (b).**
(a) (8 pts) Given the stack interface from homework 1, write Java code that will determine which value in the stack is the largest and remove and return that value while otherwise leaving the remainder of the stack in its original order.

Thus if the `stack1` contains values: (bottom 4 6 8 3 2 5 top) then a call to `findAndRemoveLargest(stack1)` would return the value 8 and leave `stack1` as follows: (bottom 4 6 3 2 5 top)

In your solution, you may only declare extra `DStacks` or scalar variables (e.g. ints, doubles – not arrays or linked lists) if needed. You may not use other methods or classes from java collections. You can assume that any `DStacks` you use will never become full and that the provided stack will not contain duplicate values. Rather than throwing an exception, if the provided stack is empty `findAndRemoveLargest(DStack)` will return the value -373.

Unlike homework 1, assume that the only implementation of `DStack` that currently exists is the class `LogStack`. `LogStack` implements all of the operations of the `DStack` interface, but does them each at a cost of O(log N). If you need to create any temporary `DStacks` in your code, you will need to create new `LogStacks`. The interface for `DStack` is given below:

```
public interface DStack {
    // Returns true if stack is empty, false otherwise.
    public boolean isEmpty();

    // Adds d to the top of the stack.
    public void push(double d);

    // Returns and removes the value on top of stack.
    // @throws EmptyStackException if stack is empty
    public double pop();

    // Returns (but does not remove) the value on top of stack.
    // @throws EmptyStackException if stack is empty
    public double peek();
}
```

Please write your code for `findAndRemoveLargest(DStack myStack)` on the next page.

(b) (3 pts) Since the only implementation of `DStack` that currently exists is the class `LogStack`, all `DStack` parameters passed to `findAndRemoveLargest(DStack myStack)` must be `LogStack`s. Thus **all** stack operations `(isEmpty, push, pop, peek)` on any stacks in your code take time O(log n) each.

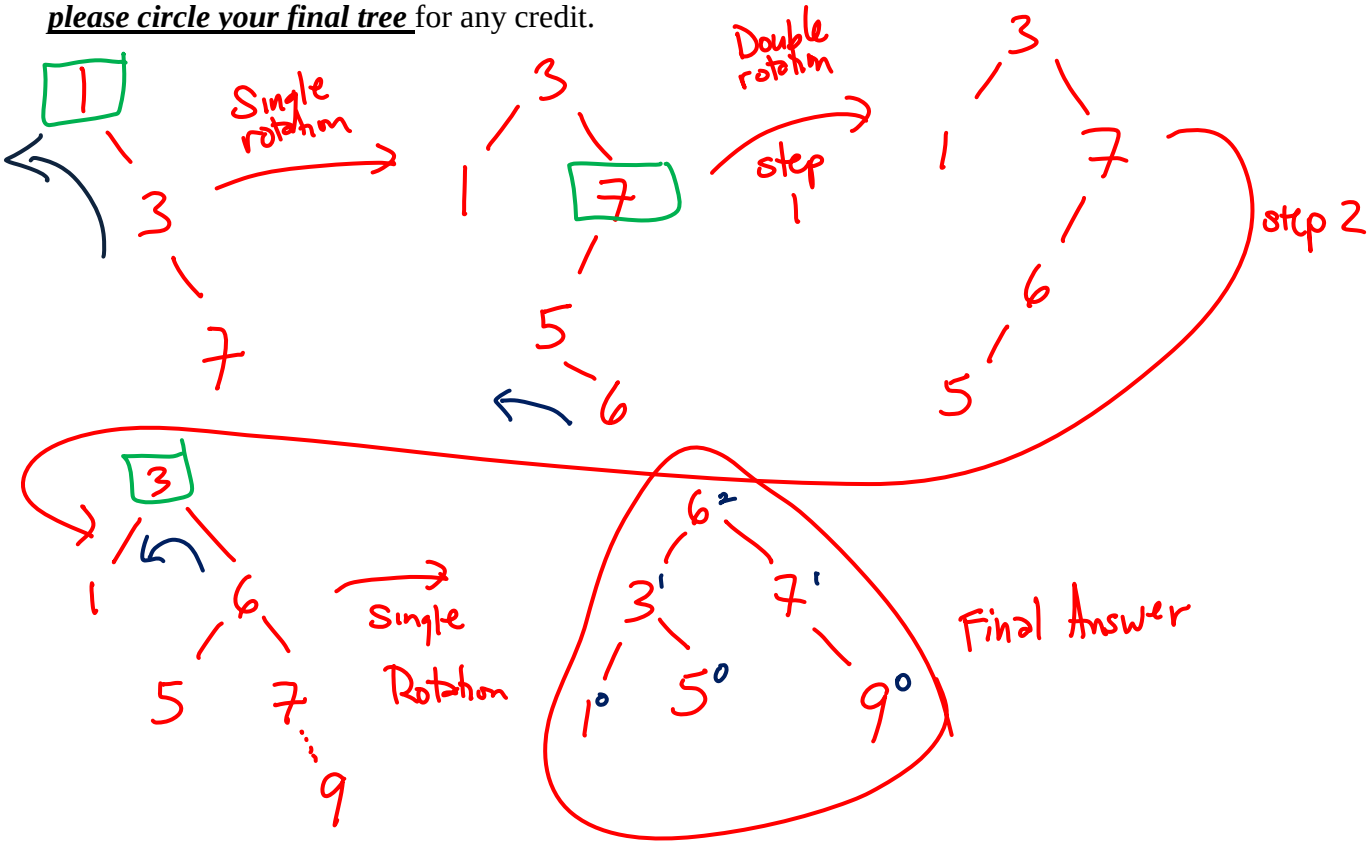
What is the worst case Big-O running time of your implementation of `findAndRemoveLargest`? For full credit give your answer in the most simplified form possible. (An explanation is not required, but could help provide partial credit. No credit if your solution is not mostly correct.)

`O(N log N)`

```
// Finds and removes the largest value in myStack.
// Returns the largest value, otherwise leaves myStack in original order.
// Returns -373 if myStack is empty.
public static double findAndRemoveLargest(DStack myStack) {
	// Add your code here:

	if (myStack.isEmpty()) return -373;
	else {
		DStack tempStack = new LogStack();
		double max = myStack.peek();
		// Find the largest value, place values on tempStack
		while (!myStack.isEmpty()) {
			double temp = myStack.pop();
			if (temp > max) max = temp;
			tempStack.push(temp);
		}
		// Put all values except max back on orig stack
		while (!tempStack.isEmpty()) {
			double temp = tempStack.pop();
			if (temp != max) myStack.push(temp);
		}
		return max;
	}
}
```

5. (6 pts) **AVL Trees** Draw the AVL tree that results from inserting the keys: 1, 3, 7, 5, 6, 9 in that order into an initially empty AVL tree. You are only required to show the final tree, although drawing intermediate trees may result in partial credit. If you draw intermediate trees, ***please circle your final tree*** for any credit.



6. (12 pts) **Algorithms & Running Time Analysis:**

- **Describe the most time-efficient way to implement the operations listed below**. Assume no duplicate values and that you can implement the operation as a member function of the class – with access to the underlying data structure, including knowing the number of values currently stored (N).
- Then, give the tightest possible upper bound for the ***worst case*** running time for each operation in terms of *N*. ****For any credit, you must explain *why* it gets this worst case running time.** You must choose your answer from the following (not listed in any particular order), each of which could be re-used (could be the answer for more than one of a) -f)).

  $O(N^2)$, $O(N^{½})$, $O(N^3 \log N)$, $O(N \log N)$, $O(N)$, $O(N^2 \log N)$, $O(N^5)$, $O(2^N)$, $O(N^3)$, $O(\log N)$, $O(1)$, $O(N^4)$, $O(N^{12})O(N^N)$, $O(N^6)$, $O(N^8)$, $O(\log \log N)$

a) Given a FIFO queue implemented as a linked list, find and remove all of the values greater than 10, leaving the rest of the queue in its original order. **Explanation:**

a)

O(N)

Since you know the size of the queue, you can loop through all N values:

```
for (int i=0; i<=N; i++) {
  temp = Q.dequeue()
  if (temp <= 10) Q.enqueue(temp)
}
```

Assuming your queue is set up like this:

Each enqueue and dequeue operation takes time O(1), the total runtime is O(N). There is no real worst case.

Alternately, since we can also consider this a member function we could traverse the linked list directly from front to back (O(N)) and for each node, if it is > 10, then adjust pointers to remove it from the list (O(1) per removal just setting pointers) also giving us running time O(N).

b) Given a binary search tree, print all of the odd values from largest to smallest. Be specific about how you would get values in order from largest to smallest.
Explanation:

b)

O(N)

```
printOdd(node root) {
  if root == null return
  printOdd(root.right)
  if (root.val % 2 == 1) print it
  printOdd(root.left)
}
```

This is essentially an inorder traversal done in reverse. Since you must visit each node once, and do constant time work at each node, overall time is O(N). There is no real "worst case".

6. (continued) **Algorithms & Running Time Analysis:**

- **Describe the most time-efficient way to implement the operations listed below**. Assume no duplicate values and that you can implement the operation as a member function of the class – with access to the underlying data structure, including knowing the number of values currently stored (N).
- Then, give the tightest possible upper bound for the ***worst case*** running time for each operation in terms of *N*. ****For any credit, you must explain *why* it gets this worst case running time.** You must choose your answer from the following (not listed in any particular order), each of which could be re-used (could be the answer for more than one of a) -f)).

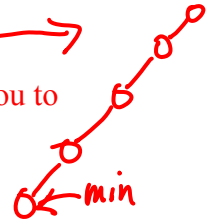
  $O(N^2)$, $O(N^{½})$, $O(N^3 \log N)$, $O(N \log N)$, $O(N)$, $O(N^2 \log N)$, $O(N^5)$, $O(2^N)$, $O(N^3)$, $O(\log N)$, $O(1)$, $O(N^4)$, $O(N^{12})O(N^N)$, $O(N^6)$, $O(N^8)$, $O(\log \log N)$

c) Given a binary search tree, find which value is the *minimum* value **and** delete it.
Explanation:

c) O(N)

To findmin in a BST, start at the root and go left until you encounter a null pointer – you have now found the minimum value. Deleting this node is easy, since it is either a leaf node (just remove it) or a node with one child (just replace node with its child) – both of which take constant time.

In the worst case you are dealing with a "linked-list tree" slanting to the left, forcing you to traverse all N nodes in order to find the minimum. Thus worst case runtime is O(N).



d) Given a FIFO queue implemented as a linked list currently containing N values, enqueue N more values so that when you are finished the queue will contain 2N values (Give the total time for enqueueing N more values). **Explanation:**

d) O(N)

Whether the queue is empty or contains N values, the cost of an enqueue operation is only constant time (back.next = new node(x), back =back.next). So doing N enqueue operations will take time O(N). There is no real "worst case".