Computer Science Foundation Exam

December 13, 2013

Section I B

COMPUTER SCIENCE

NO books, notes, or calculators may be used, and you must work entirely on your own.

SOLUTION

Question #	Max Pts	Category	Passing	Score
1	10	ANL	7	
2	10	DSN	7	
3	10	DSN	7	
4	10	ALG	7	
5	10	ALG	7	
TOTAL	50		35	

You must do all 5 problems in this section of the exam.

Problems will be graded based on the completeness of the solution steps and not graded based on the answer alone. Credit cannot be given unless all work is shown and is readable. Be complete, yet concise, and above all be neat.

1) (10pts) ANL (Algorithm Analysis)

Consider two competing ideas to calculate class rank for students on exams. Assume the possible grades on the exam are all of the integers 0 through *m*, and that there are *n* exam takers, where both *m* and *n* are positive integers.

> 1) Create an integer array of size *m+1*, where index *i* will store the number of students who scored ***i* points or less** on the exam.
>
> 2) Create an integer array of size *n*, where the array stores each student's score in sorted order.

(a) (3 pts) In implementing idea #1, what is the run-time of generating the array from the process of reading in the *n* values? Assume an efficient implementation. Provide a Big-Oh run-time in terms of *n* and *m*. Justify your answer.

The run time is O(n) (2 pts). To implement this idea, read through each piece of data, incrementing that array index. Each increment takes O(1) time and there are n of them. Then, sweep through the array a second time, iteratively adding each array element to a running sum and store this total back in each array index. This is a second loop of n iterations with O(1) work in each iteration. Adding everything, we get O(n) time. (1 pt – award this point if there's some awareness that we can efficiently create the cumulative frequency array, or if they answered $O(n^2)$ and their justification matches that run time.)

(b) (3 pts) In implementing idea #2, what is the run-time of generating the array from the process of reading in the *n* values into the array followed by a Merge Sort of the values? Assume an efficient implementation. Provide a Big-Oh run-time in terms of *n* and *m*. Justify your answer.

The run time is O(nlgn). (2 pts) Reading in the values takes O(n) time and sorting those values takes O(nlgn), the worst case time of Merge Sort. Adding, we get O(nlgn). (1 pt)

(c) (2 pts) If we are using idea #1 and we've already populated our data, what is the run-time of determining the number of exams scores less than or equal to some given score *S*? Assume an efficient implementation. Provide a Big-Oh run-time in terms of *n* and *m*. Justify your answer.

The query is precisely what is stored at index *S*, so all we have to do is return the value stored there, which we can do in O(1) time. (1 pt answer, 1 pt justification, give 1 pt to incorrect consistent responses.)

(d) (2 pts) If we are using idea #2 and we've already populated our data, what is the run-time of determining the number of exams scores less than or equal to some given score *S*? Assume an efficient implementation. Provide a Big-Oh run-time in terms of *n* and *m*. Justify your answer.

One way to solve the problem would be to do a binary search in the array for the maximal index storing a score of S or less. Though an adaptation of a regular binary search, this binary search also takes O(lg n) time. (1 pt answer, 1 pt justification, give 1 pt to incorrect consistent responses.)

2) (10 pts) DSN (Recursive Algorithms – Binary Trees)

Write a **recursive** `function averageValue` that returns the average value of all the nodes in a binary tree pointed to by its input parameter root. Assume that in each node, there is a component called `numNodes` which correctly stores the total number of nodes (including itself) that are in the subtree with it as a root. (Note: If n_1 values have an average of A_1, and n_2 values have an average of A_2, then the average if those $n_1 + n_2$ values is $\frac{n_1 A_1 + n_2 A_2}{n_1 + n_2}$.) **You are guaranteed that the pointer passed to the function will point to a non-empty tree.**

Use the following struct definition:

```
struct treeNode {
    int data;
    int numNodes;
    struct treeNode *left;
    struct treeNode *right;
};


double averageValue(struct treeNode* root) {

    // 2 pts
    if (root->left == NULL && root->right == NULL)
        return root->data;

    // 1 pt set up.
    double sum = root->data;
    int totalNodes = 1;

    // 3 pts left
    if (root->left != NULL) {
        int size = root->left->numNodes;
        totalNodes += size;
        sum += (averageValue(root->left)*size);
    }

    // 3 pts right
    if (root->right != NULL) {
        int size = root->right->numNodes;
        totalNodes += size;
        sum += (averageValue(root->right)*size);
    }

    // 1 pt return
    return sum/totalNodes;
}
```

3) (10 pts) DSN (Linked Lists)

Write a function, `delLast`, that deletes the last node in the list pointed to by the input parameter `front`. Your function should return the front of the resulting list. **If the given list is empty, make no changes and return NULL.**

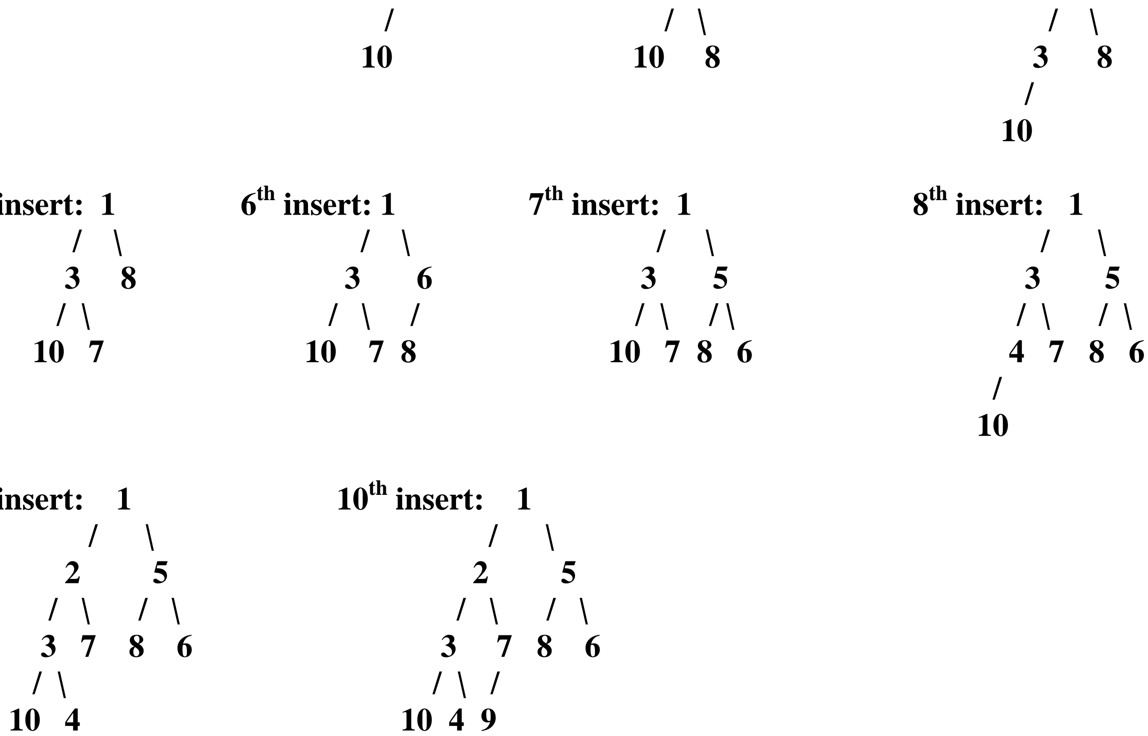
Use the struct definition provided below.

```
struct node {
    int data;
    struct node *next;
};

struct node* delLast(struct node *front) {

    // 2 pts
    if (front == NULL)
        return NULL;

    // 3 pts
    if (front->next == NULL) {
        free(front);
        return NULL;
    }

    // 4 pts
    front = delLast(front->next);

    // 1 pt
    return front;
}
```

Note: For an iterative solution, map the points shown above. 2 pts for the empty list case, 3 pts for the 1 node case, and 5 pts for the multiple node case.

4) (10 pts) ALG (Heaps)

Consider inserting the following items into a minimum heap, in this order: 10, 3, 8, 1, 7, 6, 5, 4, 2, and 9. Draw the tree representation of the heap after each insertion. Clearly label each answer by putting a box around it.



Grading: 1 pt per heap, give credit only if the structure is perfectly correct.

5) (10 pts) ALG (Sorting)

(a) (5 pts) Consider sorting the array below using Bubble Sort. Show the contents of the array, after each iteration of the outer loop. (Note: After each iteration, the maximal element iterated over should be in its correct location.)

Original	3	5	4	6	1	2
1st iteration	**3**	**4**	**5**	**1**	**2**	**6**
2nd iteration	**3**	**4**	**1**	**2**	**5**	**6**
3rd iteration	**3**	**1**	**2**	**4**	**5**	**6**
4th iteration	**1**	**2**	**3**	**4**	**5**	**6**
5th iteration	**1**	**2**	**3**	**4**	**5**	**6**

Grading: 1 pt per row, only award a point if the row is fully correct.

(b) (5 pts) In a typical Merge Sort, our base case is an array of size 1. Consider changing the base case of Merge Sort so that we attempt to see if the subarray we seek to sort is already sorted. If it is, we return immediately instead of following the recursive algorithm. In code, assuming isSorted and Merge functions were already written, it would look like this:

```
void MergeSort(int* array, int low, int high) {
    if (!isSorted(array, low, high)) {
        MergeSort(array, low, (low+high)/2);
        MergeSort(array, (low+high)/2+1, high);
        Merge(array, low, (low+high)/2, high);
    }
}
```

(b – i) (2 pts) Assuming that isSorted and Merge are written efficiently, what is the best case run time of this modified Merge Sort, in terms of *n*, the size of its input array?

It's O(n), because if we pass an already sorted array to it, the isSorted function, which will just iterate through the array once in O(n) time, will return 1/true, and the sort will be finished. (2 pts for the answer, no justification necessary)

(b – ii) (3 pts) Assuming that isSorted and Merge are written efficiently, what is the worst case run time of this modified Merge Sort, in terms of *n*, the size of its input array?

It's O(nlgn), the same as regular Merge Sort. To see this, note that by adding the call to isSorted, we are adding O(n) work to the recursive function, compared to the original. If we let T(n) be the function for the run time of this Merge Sort for an input size of n, we find that T(n) satisfies the following recurrence:

$T(n) = 2T(n/2) + O(n) + O(n)$, for the 2 recursive calls, the isSorted call and the Merge call.

Simplifying, we get $T(n) = 2T(n/2) + O(n)$, the same recurrence as regular Merge sort. It follows that its solution is the same, O(nlgn). (3 pts for the answer, no justification necessary)