Computer Science Foundation Exam

December 13, 2013

Section I A

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	DSN	7	
2	10	ANL	7	
3	10	ALG	7	
4	10	ALG	7	
5	10	ALG	7	
TOTAL	50			

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) (10 pts) DSN (Recursion)

Write a function, `oddDepthSum(node *root)`, that returns the sum of all values that lie at an odd depth in a binary tree. You may write helper functions as you see fit. For example, your function should return 20 + 30 + 13 = 63 for the following tree:

```
        10      <- depth: 0 (even)
       /  \
      20  30    <- depth: 1 (odd)
     /  \
    8    7      <- depth: 2 (even)
     \
      13        <- depth: 3 (odd)
```

```
// here is the node struct definition:

typedef struct node {
  int data;
  struct node *left, *right;
} node;
```

```
int oddDepthSum(node *root) {

     return foo(root, 0);           // 2 pts
}

int foo(node *root, int odd) {          // 1 pt

     if (root == NULL)                  // 1 pt
          return 0;

     return (odd ? root->data : 0)      //  6pts total
            + foo(root->left, !odd)     // 2 pts for each call
            + foo(root->right, !odd);   // 2 pts for the rest
}
```

Note: There are many solutions. Map the points according to the ideas in the solution above. Most students are likely to use if statements instead of the ternary operator and explicitly keep track of a depth and check its value mod 2.

2) (10 pts) ANL (Summations)

(a) (3 pts) Write a summation that represents the number of times the statement `p++` is executed in the following function:

```
int foo(int n)
{
   int i, j, p = 0;

   for (i = 1; i < n; i++)
      for (j = i; j <= i + 10; j++)
         p++;

   return p;
}
```

Solution: $\sum_{i=1}^{n-1} \sum_{j=i}^{i+10} 1$ **Grading: 1 pt inside, 1 pt inner sum, 1 pt outer sum**

(b) (5 pts) Determine a simplified, closed-form solution for your summation from part (a), in terms of *n*. **You MUST show your work.**

Solution: $\sum_{i=1}^{n-1} \sum_{j=i}^{i+10} 1 = \sum_{i=1}^{n-1} (\sum_{j=1}^{i+10} 1 - \sum_{j=1}^{i-1} 1) = \sum_{i=1}^{n-1} ((i+10) - (i-1))$

$= \sum_{i=1}^{n-1} (11) = 11 \sum_{i=1}^{n-1} 1 = 11(n-1) = 11n - 11$

Grading: 2 pts for 11, 3 pts for rest.

(c) (2 pts) Using big-oh notation, what is the runtime of the *foo()* function from part (a)?

Solution: O(n) **Grading: All or nothing**

3) (10 pts) ALG (Binary Search Trees)

(a) (3 pts) Using big-oh notation, give the best-, worst-, and average-case **runtime** of insertion into a binary search tree that already has *n* elements:

Best-case: **O(1)** **Grading: 1 pt each all or nothing**

Worst-case: **O(n)**

Average-case: **O(log n)**

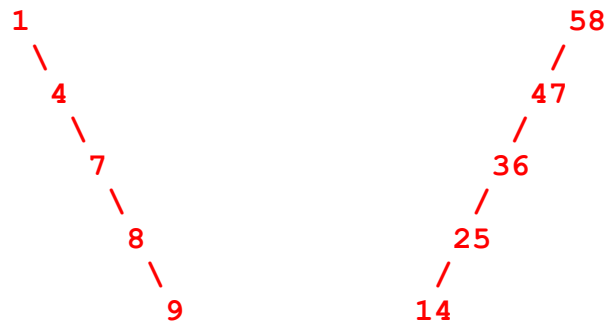
(b) (2 pts) Using big-oh notation, indicate the minimum and maximum **heights** that a binary search tree with *n* elements can have:

Minimum height: **O(log n)** **Grading: 1 pt each all or nothing**

Maximum height: **O(n)**

(c) (5 pts) Draw a binary search tree with **5 positive integers** such that inserting the integer 13 will incur the worst-case runtime for BST insertion.

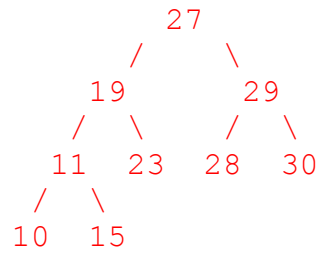
There are infinitely many correct solutions. The point is for the BST to devolve into a linked list. For example, here are two possible solutions:



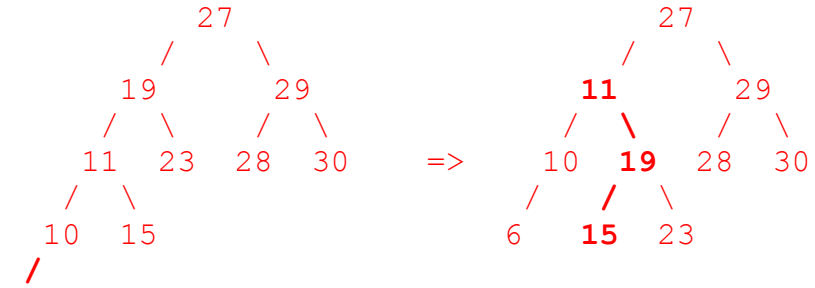
Grading: 5 pts for a correct answer, 3 pts if height is correct but 13 doesn't go all the way down, 2 pts if height is short, 0 otherwise

4) (10 pts) ALG (AVL Trees)

(a) (4 pts) Insert the value 6 into the following AVL tree. Clearly show where the node is inserted initially and what the tree looks like after **each** rotation that takes place (if any).

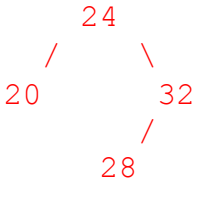


Solution: 6 is inserted as the left child of 10. This causes a single right rotation at node 19. During the rotation, 11 moves up to take the place of 19. Node 19 moves down, and the old right child of 11 becomes the new left child of 19.



Grading: 1 pt for keeping 27
1 pt for 11, 19 and 15

(b) (4 pts) List all the integer values that would cause a **double rotation** if inserted into the following AVL tree. If there are no such values, say so. (Assume we do not allow the insertion of duplicate values into our AVL trees.)



Solution: Inserting 29, 30, or 31 would cause a double rotation.
No other values would cause a double rotation.

// 1 pt for each correct answer 1 pt for not
// listing any extra

(c) (2 pts) In big-oh notation, what is the **worst-case** runtime of inserting a single element into an AVL tree that already has *n* elements?

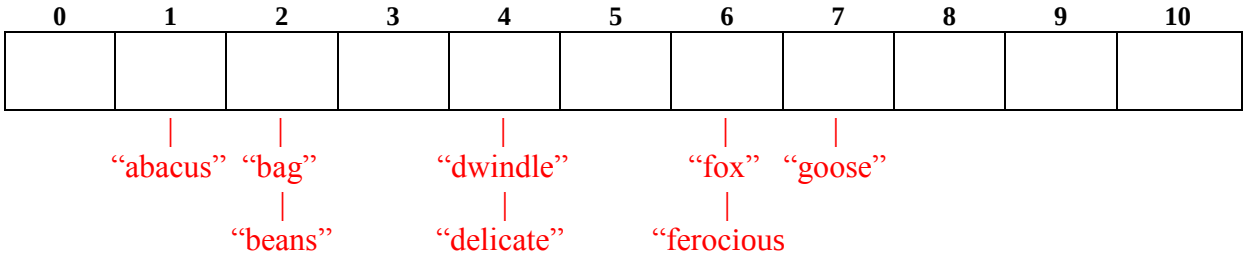
Solution: O(log n) **Grading: 2 pts all or nothing**

5) (10 pts) ALG (Hash Tables)

(a) (4 pts) Insert the following elements into the hash table below using **separate chaining** to resolve collisions. The table is of size 11, so assume that the hash value for a given input, *key*, is determined by calling hash(*key*, 11).

```
int hash(char *s, int table_size) {
   if (s == NULL || s[0] == '\0') return 0;
   else return ((int)(tolower(s[0]) - 'a') + 1) % table_size;
}
```

Keys to insert: "abacus", "dwindle", "fox", "goose", "bag", "beans", "ferocious", "delicate"



Grading: ½ pt for each, round down

(b) In general, what is the **worst-case** big-oh runtime for insertion into a hash table that already contains *n* elements? *Briefly* describe the specific situation that leads to this worst-case performance.

Solution: The worst case is O(n), which happens when there are excessive collisions (possibly as a result of a poor hash function that does not produce evenly distributed hash values, or because of clusters of data that form while using linear or quadratic probing). **Grading: 2 pts ans, 1 pt reason**

(c) Suppose we want to use this hash function to insert a much larger set of strings into our hash table (some arbitrarily large number of strings, *n*), and so we expand the hash table to be length O(n). Is the hash function in this problem a good hash function for that purpose? Why or why not?

Solution: No. The hash function is terrible because it will only ever produce 27 unique hash values, even if we have a huge table. That will lead to lots of collisions for large *n*. **Grading: 1 pt for answer, 2 pts for reason**