Computer Science Foundation Exam

January 13, 2018

Section I A

DATA STRUCTURES

SOLUTIONS

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

Question #	Max Pts	Category	Passing	Score
1	10	DSN	7	
2	5	DSN	3	
3	10	ALG	7	
TOTAL	25		17	

You must do all 3 problems in this section of the exam.

Problems will be graded based on the completeness of the solution steps and <u>not</u> 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 <u>*be neat*</u>. For each coding question, assume that all of the necessary includes (stdlib, stdio, math, string) for that particular question have been made.

1) (10 pts) DSN (Dynamic Memory Management in C)

The struct, `dataTOD`, shown below, is used to collect data from different devices connected to the CPU. Every time the data is updated a new buffer containing the structure's data is created and populated.

```
typedef struct dataTOD{
    int seconds;       // seconds since midnight
    double data;       // data sample
    char * dataName;   // data name (optional)
} dataTOD;
```

(a) (8 pts) Write the code necessary to create and initialize the members of `dataTOD` in a function named `init_dataTOD` that returns a pointer to the newly created buffer. Return a NULL in the event a buffer cannot be created. Otherwise, set the seconds and data values according to the corresponding input parameters to `init_dataTOD`, dynamically allocate the proper space for dataName and then copy the contents of name into it (not a pointer copy) and a return a pointer to the newly created struct.

```
dataTOD * init_dataTOD(int sec, double val, char* name){

    dataTOD * tDta = malloc(sizeof(dataTOD));
    if (tDta == NULL)
        return NULL;
    tDta->seconds = sec;
    tDta->data = val;
    tDta->dataName = malloc((strlen(name)+1)*sizeof(char));
    strcpy(tDta->dataName, name);
    return tDta;
}
```

Grading: 1 point for each line shown above. Assign partial if necessary but assign a whole number of points.

(b) (2 pts) Complete the function below so that it frees all the dynamically allocated memory pointed to by its formal parameter zapThis. You may assume that the pointer itself is pointing to a valid struct and its dataName pointer is pointing to a dynamically allocated character array.

```
void free_dataTOD(dataTOD *zapThis){
    free(zapThis->dataName);
    free(zapThis);
}
```

Grading: 1 pt each line

2) (5 pts) DSN (Linked Lists)

Given the linked list structure named `node`, defined in lines 1 through 4, and the function named `eFunction` defined in lines 6 through 14, answer the questions below.

```
 1 typedef struct node {
 2         int data;
 3         struct node * next;
 4 } node;
 5
 6 node* eFunction(node* aNode){
 7    if(aNode == NULL) return NULL;
 8    if(aNode->next == NULL) return aNode;
 9
10    node* rest = eFunction(aNode->next);
11    aNode->next->next = aNode;
12    aNode->next = NULL;
13    return rest;
14 }
```

(a) (1 pt) Is this function recursive? (Circle the correct answer below.)

YES (1 pt) NO

(b) (2 pts) What does the function eFunction do, in general to the list pointed to by its formal parameter, aNode?

This function reverses the list originally pointed to by aNode and returns a pointer to the new front of the list. (**Grading:** 1 pt for reverse, 1 pt for return pointer to reversed list.)

(c) (2 pts) What important task does line 12 perform?

The last node in a linked list must have its next pointer point to NULL. That is how most linked list functions detect the end of the list. Line 12 does this since aNode ends up point to the last node in the list. After the reversal is complete, it's necessary to make sure that the next pointer of the last node in the resulting list is pointing to NULL because before line 12 it's not. (It's pointing to the second node in the original list, which is the last node in the list pointed to by rest.)

Grading: Most of this detail is unnecessary. Full credit for noting that the line sets the last node's next pointer of the resulting list to NULL. Give partial as needed.

3) (10 pts) ALG (Stacks) Consider evaluating a postfix expression that only contained ***positive*** integer operands and the addition and subtraction operators. (Thus, there are no issues with order of operations!) Write a function that evalulates such an expression. To make this question easier, assume that your function takes an array of integers, `expr`, storing the expression and the length of that array, `len`. In the array of integers, all positive integers are operands while -1 represents an addition sign and -2 represents a subtraction sign. Assume that you have a stack at your disposal with the following function signatures. Furthermore, assume that the input expression is a valid postfix expression, so you don't have to ever check if you are attempting to pop an empty stack. Complete the evaluate function below.

```
void init(stack* s); // Initializes the stack pointed to by s.
void push(stack* s, int item); // Pushes item onto the stack pointed
                               // to by s.
int pop(stack* s); // Pops and returns the top value from the stack
                   // pointed to by s.

int eval(int* expr, int len) {

    stack s;
    init(&s);
    int i;

    for (i=0; i<len; i++) {              // 1 pt
        if (expr[i] > 0)                 // 1 pt
            push(&s, expr[i]);           // 1 pt
        else {
            int op2 = pop(&s);           // 1 pt
            int op1 = pop(&s);           // 1 pt
            if (expr[i] == -1)           // 1 pt
                push(&s, op1+op2);       // 1 pt
            else
                push(&s, op1-op2);       // 2 pts (1 pt for order)
        }
    }

    return pop(&s);                      // 1 pt
}
```

Computer Science Foundation Exam

January 13, 2018

Section I B

DATA STRUCTURES

SOLUTIONS

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

Question #	Max Pts	Category	Passing	Score
1	10	DSN	7	
2	5	ALG	3	
3	10	ALG	7	
TOTAL	25		17	

You must do all 3 problems in this section of the exam.

Problems will be graded based on the completeness of the solution steps and <u>not</u> 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 <u>*be neat*</u>. For each coding question, assume that all of the necessary includes (stdlib, stdio, math, string) for that particular question have been made.

1) (10 pts) DSN (Binary Search Trees)

Write a ***recursive*** function to find the ***leaf node*** in a binary search tree storing the minimum value. (Thus, of all leaf nodes in the binary search tree, the function must return a pointer to the one that stores the smallest value.) If the pointer passed to the function is NULL (empty tree), the function should return NULL.

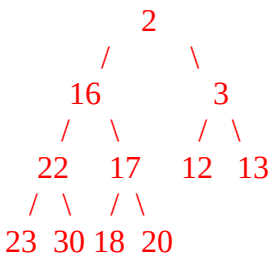
```
typedef struct bstNode {
    int data;
    struct bstNode *left;
    struct bstNode *right;
} bstNode;

bstNode* find_min_leaf(bstNode* root) {

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

    if (root->left == NULL && root->right == NULL)// 2 pts
        return root;                               // 1 pt

    if (root->left != NULL)                    // 1 pt
        return find_min_left(root->left);      // 2 pts

    return find_min_left(root->right);         // 2 pts

}
```

2) (5 pts) ALG (Binary Heaps)

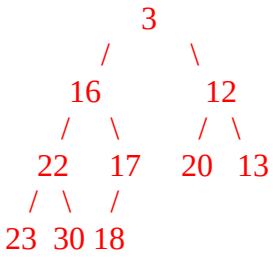
The array below stores a minimum binary heap. Draw the tree version of the corresponding binary heap. Then, remove the minimum value and show the resulting heap, in tree form. (Note: Index 0 isn't shown because index 1 stores the value at the root/top of heap.)

Index	1	2	3	4	5	6	7	8	9	10	11
Value	2	16	3	22	17	12	13	23	30	18	20

Here is the initial heap, in tree form:



When we delete the minimum, 2, stored at the top, 20, the value in the "last" location replaces it (to maintain the structural integrity of the heap.) From there, we percolate 20 down, swapping it with 3, and then 12 to get the resulting tree:



Grading: 2 pts for correct drawing, 1 pt is something minor is off, 0 otherwise.
1 pt if structural location of 20 is removed, 1 pt for incorrect percolateDown,
2 pts for correct percolateDown. (If the drawing is significantly wrong, don't give any credit for the second part. If it's slightly wrong, map points as best as possible.)

3) (10 pts) ALG (AVL Trees)

Show the result of inserting the following values into an initially empty AVL tree:

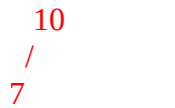
10, 7, 3, 22, 16, 13, 5, 18, 20 and 19.

Draw a box around your result ***after each insertion.***

The resulting trees are as follows:

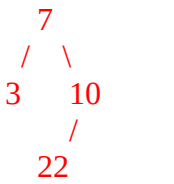
Tree 1:

Tree 2:

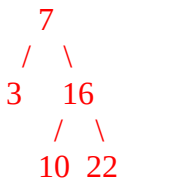


Tree 3:

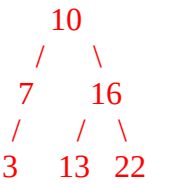
Tree 4:



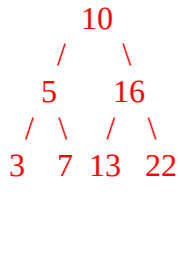
Tree 5:



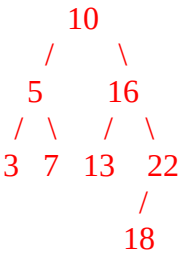
Tree 6:



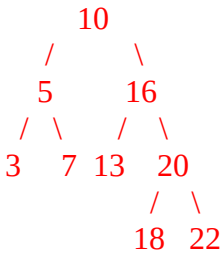
Tree 7:



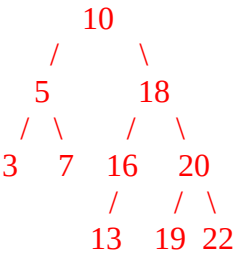
Tree 8:



Tree 9:



Tree 10:



Grading: 1 pt per tree, try to judge each insertion based on their previous tree (so they can get a point even if their answer doesn't match as long as it is correct based on their previous tree.)

Computer Science Foundation Exam

January 13, 2018

Section II A

ALGORITHMS AND ANALYSIS TOOLS

SOLUTION

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

Question #	Max Pts	Category	Passing	Score
1	10	ANL	7	
2	5	ANL	3	
3	10	ANL	7	
TOTAL	25		17	

You must do all 3 problems in this section of the exam.

Problems will be graded based on the completeness of the solution steps and <u>not</u> 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 <u>be neat</u>. For each coding question, assume that all of the necessary includes (stdlib, stdio, math, string) for that particular question have been made.

1) (10 pts) ANL (Algorithm Analysis)

With proof, determine the Big-Oh run time of the function, f, below, in terms of the input parameter n:

```
int f(int array[], int n) {

    int i, t = 0, a = 0, b = n-1;
    while (a < b) {

        for (i=a; i<=b; i++)
            t += array[i];

        if (array[a] < array[(a+b)/2])
            b = (a+b)/2-1;
        else
            a = (a+b)/2+1;

    }
    return t;
}
```

In your work, you may use the following result: $\sum_{i=0}^{\infty}(\frac{1}{2})^i = 2$.

The main loop in the function is a while loop. The while loop is controlled by a and b, which initially start as the low and high indexes into array (assuming its of length n). The if else statement essentially resets either a or b to be halfway between the two. In essence, the difference between a and b is being divided by 2, roughly, at each iteration of the while loop. Thus, it follows that the while loop will run roughly $\log_2 n$ times. It turns out that knowing this isn't critical to the Big-Oh analysis.

Now, at each iteration of the while loop, we see that a for loop is run and that the assignment statement inside the loop runs b - a + 1 times, which essentially equals the difference between a and b.

Clearly, the first time, this difference is n (with the +1 factor). The following time, this difference will be roughly $\frac{n}{2}$, since the difference between a and b gets divided by 2 with each while loop iteration. On the third while loop iteration, the inner for loop will run roughly $\frac{n}{4}$ times. In general, on the k^{th} iteration of the while loop, the inner for loop runs no more than $\frac{n}{2^{k-1}}$ times. We can claim it's an upper bound because a and b are reset to be slightly less than half of the range with the -1 and +1 respectively. It follows that the run time of the code is less than or equal to

$$\sum_{k=1}^{\infty} {}^{n}/_{2^{k-1}} = \sum_{i=0}^{\infty} {}^{n}/_{2^{i}} = n\sum_{i=0}^{\infty}(\tfrac{1}{2})^i = 2n = O(n)$$

Grading: 3 pts for realizing that the difference between a and b is being divided by 2 through each loop iteration, 3 pts for determining that the relevant sum has the pattern n + n/2 + n/4 + … 4 pts for setting up the sum, completing it and getting a Big-Oh bound. Max 7 pts for a valid O(nlgn) justification.

2) (5 pts) ANL (Algorithm Analysis)

An algorithm processing an array of size n runs in $O(n^3)$ time. For an array of size 500 the algorithm processes the array in 200 ms. How long would it be expected for the algorithm to take when processing an array of size 1,500? Please express your answer in ***seconds.***

Let the algorithm with input size n have a runtime of $T(n) = cn^3$, for some constant c. Using the given information we have:

$$T(500) = c(500)^3 = 200ms$$
$$c = \frac{200}{500^3}ms$$

Now, we must find T(1500):

$$T(1500) = c(1500)^3 = \frac{200}{500^3}ms \times (1500)^3 = (\frac{1500}{500})^3 \times 200ms = 3^3 \times 200ms = 5400ms = 5.4s$$

Thus, our final answer is **5.4 seconds.**

Grading: 1 pt to set up the initial equation for c, 1 pt to solve for c, 2 pts to get answer in ms, 1 pt to convert to seconds. Give partial credit for the 2 pts if the setup is correct but some algebra issue occurred. Also, give full credit if the ratio method (which is also valid) is used instead of this method. Map points accordingly if some error is made using that method.

3) (10 pts) ANL (Summations and Recurrence Relations)

Using the iteration technique, find a tight Big-Oh bound for the recurrence relation defined below:

$$T(n) = 3T\left(\frac{n}{2}\right) + n^2, \text{ for } n > 1$$
$$T(1) = 1$$

Hint: You may use the fact that $\sum_{i=0}^{\infty}(\frac{3}{4})^i = 4$ and that $3^{log_2 n} = n^{log_2 3}$, and that $log_2 3 < 2$.

Iterate the given recurrence two more times:

$$T(n) = 3T\left(\frac{n}{2}\right) + n^2$$
$$T(n) = 3(3T\left(\frac{n}{4}\right) + (\frac{n}{2})^2) + n^2$$
$$T(n) = 9T\left(\frac{n}{4}\right) + \frac{3n^2}{4} + n^2$$
$$T(n) = 9T\left(\frac{n}{4}\right) + n^2(1 + \frac{3}{4})$$
$$T(n) = 9(3T\left(\frac{n}{8}\right) + (\frac{n}{4})^2) + n^2(1 + \frac{3}{4})$$
$$T(n) = 27T\left(\frac{n}{8}\right) + \frac{9n^2}{16} + n^2(1 + \frac{3}{4})$$
$$T(n) = 27T\left(\frac{n}{8}\right) + n^2(1 + \frac{3}{4} + \frac{9}{16})$$

In general, after the k[th] iteration, we get the recurrence

$$T(n) = 3^k T\left(\frac{n}{2^k}\right) + n^2(\sum_{i=0}^{k-1}(\frac{3}{4})^i)$$

To solve the recurrence, find k such that $\frac{n}{2^k} = 1$. This occurs when $n = 2^k$ and $k = log_2 n$. Plug into the equation above for this value of k to get:

$$T(n) = 3^{logn}T(1) + n^2\left(\sum_{i=0}^{k-1}\left(\frac{3}{4}\right)^i\right) \leq 3^{logn} + n^2\left(\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i\right) = n^{log_2 3} + 4n^2 = O(n^2)$$

Grading: 1 pt for copying recurrence, 1 pt for getting 2[nd] iteration, 2 pts for getting third iteration (in any form), 3 pts for kth iteration, 1 pt for what to plug in form, 2 pts to complete the problem. (Be somewhat generous as this is probably the hardest problem on the exam.)

Computer Science Foundation Exam

January 13, 2018

Section II B

ALGORITHMS AND ANALYSIS TOOLS

SOLUTION

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

Question #	Max Pts	Category	Passing	Score
1	10	DSN	7	
2	5	ALG	3	
3	10	DSN	7	
TOTAL	25		17	

You must do all 3 problems in this section of the exam.

Problems will be graded based on the completeness of the solution steps and <u>not</u> 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 <u>be neat</u>.

1) (10 pts) DSN (Recursive Coding)

Write an ***efficient recursive*** function that takes in a ***sorted*** array `numbers`, two integers, `low` and `high`, representing indexes into the array, and another integer, `value`, and returns the index in the array where `value` is found in the array in between index `low` and `high`, inclusive. If `value` is NOT found in the array in between indexes `low` and `high`, inclusive, then the function should return -1.

```
int search(int numbers[], int low, int high, int value) {

    if (low > high) return -1;

    int mid = (low+high)/2;

    if (value > numbers[mid])
        return search(numbers, mid+1, high, value);
    else if (value < numbers[mid])
        return search(numbers, low, mid-1, value);
    else
        return mid;
}
```

Grading: 2 pts for return -1 base case
3 pts for going halfway in between search range
2 pts for case going to the right
2 pts for case going to the left
1 pt for base case returning mid
Max grade of 7 for linear recursive solution (no pts for going halfway…)
Max grade of 3 pts for non-recursive solution, regardless of runtime.
Max grade of 6 if recursive structure is correct but recursive call(s) are missing

2) (5 pts) ALG (Sorting)

(a) (3 pts) Explain why, in the worst case, Quick Sort runs more slowly than Merge Sort.

In the worst case for Quick Sort, every time the array get split into two sides, if the split is extremely unequal (0 items on one size and all n-1 items except the partition element on the other side), worst case behavior occurs because there are n nested recursive calls on arrays of size n, n-1, n-2, and so forth.

In Merge Sort, it's guaranteed that the recursive calls always split into two arrays of roughly equal size. In general, the more equal the split between the two recursive calls is, the better the overall run-time will be. Because the Merge Sort split is essentially fixed, its worst case run time is near equal to its average case run time. But for Quick Sort, since this split at each level of recursion can be arbitrarily unequal, in the worst case where it's extremely unequal, the sort performs worse than Merge Sort.

Grading: The amount of detail above isn't necessary. Full credit to any response that recognizes that when making two recursive calls it's better to split the input array equally and that Merge Sort guarantees this but for Quick Sort this doesn't happen in the worst case.

Award partial credit as you see fit.

(b) (2 pts) In practice, Quick Sort runs slightly faster than Merge Sort. This is because the partition function can be run "in place" while the merge function can not. More clearly explain what it means to run the partition function "in place".

To run the partition function in place means that the function doesn't have to allocate significant extra memory other than the original array to sort that is passed to it. In particular, only a single temporary extra variable is needed to perform swapping (along with the usual loop index variables). Otherwise, most of the work occurs within the already allocated memory of the array passed to the partition function.

The merge function allocates a new array such that values from the original array are copied into the newly allocated array, then copied back to the original array. Thus, this function doesn't run in place as it routinely allocates a linear amount of memory (in the size of the arrays its merging) to perform its tasks. This runs slower in practice because of the extra copy back step, even though Merge Sort splits its data in the recursive step in a more equitable (and better) fashion.

Grading: Again, the amount of detail written above isn't necessary. Give full credit for any response that simply says that "in place" means performing the task without extra memory, (or a constant amount of extra memory.)

Award partial credit as you see fit.

3) (10 pts) DSN (Backtracking)

Consider the problem of placing 8 kings on an 8 x 8 chessboard, so that no two of the kings can attack each other ***AND no two kings are on the same row or column***. (Recall that a King can move one space in each of the eight possible directions of movement: up, down, left, right or any of the four diagonals.) Complete the code skeleton below so that it prints out each solution to the 8 Kings problem. (Note: assume that the function print, which isn't included, prints out the solution that corresponds to a particular permutation of kings. For example, the permutation {2, 4, 6, 1, 3, 5, 7, 0} represents kings at the following locations (0, 2), (1, 4), (2, 6), (3, 1), (4, 3), (5, 5), (6, 7), and (7, 0).)

```
#include <stdio.h>
#include <math.h>
#define SIZE 8

void go(int perm[], int k, int used[]);
void print(int perm[]);

int main() {
    int perm[SIZE];
    int used[SIZE];
    int i;
    for (i=0; i<SIZE; i++) used[i] = 0;
    go(perm, 0, used);
    return 0;
}

void go(int perm[], int k, int used[]) {

    if ( k == SIZE ) {              //(1 pt)
        print(perm);
        return;
    }

    int i;
    for (i=0; i<SIZE; i++) {

        if ( k > 0 && abs(i-perm[k-1]) <= 1 ) continue; //(4 pts)

        if ( used[i] ) continue;          //(1 pt)

        perm[k] = i ;           //(1 pt)
        used[ i ] = 1 ;         //(1 pt)

        go(perm, k+1 , used); //(1 pt)

        used[ i ] = 0 ; //(1 pt)
    }
}
```