Computer Science Foundation Exam

January 11, 2020

Section I A

DATA STRUCTURES

SOLUTION

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

Name: _____

UCFID: _____

NID: _____

Question #	Max Pts	Category	Score
1	10	DSN	
2	5	ALG	
3	10	DSN	
TOTAL	25	----	

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)

Suppose we have an array to store the TV shows we wanted to watch over break. Now that the break is over, we have watched all the shows and we need to delete our list. Our array is an array of structures that contains the name of each show and the number of seasons to watch for that show. The name of the show is a dynamically allocated string to support the different lengths of show names. Write a function called delete_show_list that will take in the show array as well as the length of that array, and free all the memory space that the array previously took up. Your function should take in 2 parameters: the array called show_list and the length of that array, length. It should free all the dynamically allocated memory associated with the list and return NULL, to indicate that the list has been deleted.

```
struct tv_show {
      char *show_name;
      int number_of_seasons;
};

struct tv_show * delete_show_list (struct tv_show *show_list, int length) {


    int i;

    for(i = 0; i < length; i++)          // 2 pts
        free(show_list[i].show_name);    // 3 pts

    free(show_list);                     // 3 pts

    return NULL;                         // 2 pts









}
```

2) (5 pts) ALG (Linked Lists)

Suppose we have a linked list implemented with the structure below. We also have a function that takes in the head of the list and the current number of nodes in the list.

```
typedef struct node {
    int num;
    struct node* next;
} node;

int whatDoesItDo (node * head, int size) {
     node * current = head;
     node * other;

     if (size < 2)
          return size;

     other = head->next;

     while (current != NULL) {
          current->next = other->next;
          free(other);
          current = current->next;
          size--;
          if(current != NULL && current->next !=NULL) {
               current = current->next;
               other = current->next;
          }
     }

     return size;
}
```

If we call what DoesItDo(head, 8) on the following list, show the list after the function has finished and state the return value.

head -> 3 -> 8 -> 12 -> 5 -> 1 -> 7 -> 19 -> 2

Picture of List Pointed to by head After Function Call:

Head -> 3 -> 12 -> 5 -> 7 -> 19

Function Return Value: 5

Grading: 2 pts for return value (all or nothing), 3 pts for list, give 3 pts if correct, give 2 pts if off by 1 item, 1 pt if off by 2 items, 0 otherwise

3) (10 pts) DSN (Stacks)

Suppose we have implemented a stack using a linked list. The structure of each node of the linked list is shown below. The stack structure contains a pointer to the head of a linked list and an integer, size, to indicate how many items are on the stack.

```
typedef struct node {
    int num;
    struct node* next;
} node;

typedef struct Stack {
    struct node *top;
    int size;
} stack;
```

Write a function that will pop off the contents of the input stack and push them onto a newly created stack, returning a pointer to the newly created stack. In effect, your function should reverse the order of the items in the original stack, placing them in a new stack. Assume you have access to all of the usual stack functions. Assume that when you push an item onto the stack, its size automatically gets updated by the push function. Similarly for pop, size gets updated appropriately when you pop an item from a stack. Do NOT call pop or peek on an empty stack.

```
void push(stack *s, int number); // Pushes number onto stack.
int pop(stack *s);  // Pops value at top of stack, and returns it.
int peek(stack *s); // Returns value at top of stack.
int isEmpty(stack *s); // Returns 1 iff the stack is empty.


stack* reverseStack(stack* s) {

    stack *newS = malloc(sizeof(stack));

    news->size = 0;            // 2 pts
    newS->top = NULL;          // 2 pts

    while(!isEmpty(s))         // 3 pts
        push(newS, pop(s));    // 3 pts




    return newS;
}
```

Computer Science Foundation Exam

January 11, 2020

Section I B

DATA STRUCTURES

SOLUTION

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

Name: _____

UCFID: _____

NID: _____

Question #	Max Pts	Category	Score
1	10	DSN	
2	5	ALG	
3	10	DSN	
TOTAL	25	---	

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 Trees)

Write a function named *fsl()* (which stands for "**f**ind **s**mallest **l**eaf") that takes a pointer to the root of a binary tree as its only argument and returns the value of the smallest **leaf** node in the tree. Note that the tree passed to your function will **not** necessarily be a binary **search** tree. If the pointer root is NULL, fsl should return INT_MAX, which is defined below.

You **cannot** write any helper functions for this problem. You must complete all of your work in a single function. The function signature and node struct are given below.

```
#define INT_MAX 2147483647

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

int fsl(node *root) {

   int l_min;
   int r_min;

   if (root == NULL)     // 2 pts: checking for NULL as base case
      return INT_MAX;    // 1 pt

   if (root->left == NULL && root->right == NULL)  // 2 pts: identify leaf
      return root->data;                           // 1 pt: correct return
                                                   //       value when leaf
                                                   //       is encountered

   l_min = fsl(root->left);       // 2 pts: correct recursive calls (give
   r_min = fsl(root->right);      // only 1 pt here if only one recursive
                                  // call)

   return (l_min < r_min) ? l_min : r_min);  // 2 pts: returning min of
                                             // these two values

}
```

2) (5 pts) ALG (Hash Tables)

Consider the following hash function, and then answer the questions that follow:

```
// This function assumes str is non-NULL and non-empty.
int hash(char *str) {

   int index = strlen(str) - 1;

   // Note: This converts letters on the range 'a' through 'z' or
   // 'A' through 'Z' to integers on the range 0 through 25.
   // For example: 'a' -> 0, 'b' -> 1, ..., 'z' -> 25.
   return tolower(str[index]) - 'a';
}
```

a) (2 pts) Give the hash code produced for each of the following strings:

hash("Not") = **19**

hash("Know") = **22**

hash("Bright") = **19**

hash("Moon") = **13**

hash("Now") = **22**

Points are based on number of correct answers:

0 or 1 correct answers -> 0 out of 2 points

2, 3, or 4 correct answers -> 1 out of 2 points

5 correct answers -> 2 out of 2 points

b) (3 pts) Using the hash values above, insert the strings (one by one, in the order given above) into the following hash table. Use **quadratic probing** to resolve any collisions. Note that there is a standard technique for dealing with hash values that exceed the length of a table (e.g., values that exceed 9 in the case of this particular table), and it's up to you to use that technique here.

Note: The length of the hash table is **10**.

0	1	2	3	4	5	6	7	8	9
Bright		**Know**	**Moon**			**Now**			**Not**

Grading: 1 pt for "Now" being in correct spot, 2 pts for **all** other strings being in correct spots.

3) (10 pts) DSN (Tries)

It's often useful to know how many words start with a particular prefix. Given a trie that stores a dictionary of valid words (**lowercase letters only**) as well as a prefix string, write a **non-recursive** function that calculates the number of words that begin with that prefix. To aid you in your solution, the struct that stores a trie node will not only store whether or not that node represents a word or not, but it will **also** store the total number of words stored within that subtree of the trie in a variable called numwords. You may assume that the TrieNode pointer passed to the function represents the root of the whole trie storing the dictionary of words. You may assume that root is NOT NULL and prefix has at least one lowercase letter in it.

```
#include <string.h>

typedef struct TrieNode {
    struct TrieNode *children[26];
    int flag; // 1 if the string is in the trie, 0 otherwise
    int numwords; // the total # of words stored in this sub-trie.
} TrieNode;

int numWordsWithPrefix(TrieNode* root, char* prefix) {

    // 1 pt var declarations.
    int i, len = strlen(prefix);

    // 1 pt loop
    for (i=0; i<len; i++) {

        // 3 pts NULL check
        if (root->children[prefix[i]-'a'] == NULL)

            // 1 pt return for this case.
            return 0;

        // 3 pts advancing pointer down trie.
        root = root->children[prefix[i]-'a'];
    }

    // 1 pt return value.
    return root->numwords;

}
```

Computer Science Foundation Exam

January 11, 2020

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	Score
1	5	ANL	
2	10	ANL	
3	10	ANL	
TOTAL	25		

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.h, stdio.h, math.h, string.h) for that particular question have been made.

1) (5 pts) ANL (Algorithm Analysis)

What is the best and worst case runtime for the following algorithm, in terms of the input parameter n? Give a brief explanation for your answers.

```
int foo(int * arr, int n){

   if (n == 0)
      return 0;

   int j = 0, i;

   for (i = 0; i < n; i++)
      if (arr[i] > arr[j])
         j = i;

   int nLen = n - j - 1;
   return arr[j] + foo(arr + j + 1, nLen);
}
```

Best Case

The for loop runs and sets j = n - 1, which means that nLen gets set to 0. In this case, the subsequent recursive call will immediately return 0 and the original recursive call will return the value of the last array element. The run time in this case is **O(n),** since the entirety of the execution includes one for loop that runs n times and a few other simple statements. From a conceptual standpoint, the for loop identifies the index in between 0 and n-1 that stores the largest value within that range.

Worst Case

The worst case is when the array is sorted in reverse order. Every call eliminates only 1 value at the cost of n operations. The total runtime becomes **$O(n^2)$.**

Grading: 2 pts for each answer, 1 pt for all of the explanation.

2) (10 pts) ANL (Algorithm Analysis)

A backtracking solution took O(n(k^n)) time where n is the number of decisions, and k was the number of options for each decision. With n of 20 and k of 1 the time it took was approximately 10 seconds. What is the expected time required for an input of 10 decisions (n=10) where each decision has 2 options (k=2) in **seconds**?

The runtime is in seconds can be expressed as cn(k^n) where c is some constant. We can find the c by plugging in n=20 and k=1 and setting the results to 10. We find that

$$10s = c20(1^{20})$$
$$\frac{10s}{20(1)} = c$$
$$c = .5s$$

To solve for the question we plug in n=10 and k=2.

$$Answer = (.5s)10(2^{10})$$
$$= 5s(1024)$$
$$= 5120s$$

Grading:
Find c, 4 pts.
Plugging in 10 and 2, 4 pts.
Correct answer, 2 pts.

3) (10 pts) ANL (Recurrence Relations)

Use the iteration technique to solve the following recurrence relation in terms of n:

$$T(n) = 2T(n/2) + 1, for\ all\ integers\ n > 1$$
$$T(1) = 1$$

Find a tight Big-Oh answer.

$$T(n) = 2T\left(\frac{n}{2}\right) + 1$$
$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + 1$$
$$T(n) = 2\left(2T\left(\frac{n}{4}\right) + 1\right) + 1$$
$$T(n) = 4T\left(\frac{n}{4}\right) + 2 + 1$$
$$T(n) = 4T\left(\frac{n}{4}\right) + 3$$
$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + 1$$
$$T(n) = 4\left(2T(\frac{n}{8}) + 1\right) + 3$$
$$T(n) = 8T(\frac{n}{8}) + 4 + 3$$
$$T(n) = 8T(\frac{n}{8}) + 7$$

Based on these three iterations, we see that after k iterations, the recurrence is

$$T(n) = 2^k T(\frac{n}{2^k}) + (2^k - 1)$$

Plug in the value of k such that $\frac{n}{2^k} = 1$ to this recurrence. This means that $2^k = n$. Substituting, we get:

$$T(n) = nT(1) + (n - 1)$$
$$T(n) = n + (n - 1)$$
$$T(n) = 2n - 1$$

It follows that T(n) = O(n).

Grading: 2 pts for iteration with T(n/4), 2 pts for T(n/8). 2 pts for general expression after k iterations, 1 pt for the value to plug in for k. 3 pts to finish the problem.

Computer Science Foundation Exam

January 11, 2020

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	Score
1	10	DSN	
2	10	DSN	
3	5	ALG	
TOTAL	25		

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 (Recursive Coding)

Model an area of land as a two dimensional grid of integers, where each integer represents the elevation of that portion of land. Water can only flow from a grid square of higher elevation to lower elevation in one of the four cardinal directions (north, south, east and west). Complete the ***recursive*** function below that takes in a 2D array of integers storing the elevation levels of each portion of land, another 2D array of integers (storing 0 or 1 in each entry) representing which grid squares have been flooded with water (1 for flooded, 0 for not flooded), as well as the current row and column value of a grid square that just flooded, and *marks the current* and *all subsequent squares* that will get flooded as a result of the water at the given location. Once a square is flooded it remains in that state. An inbounds function and DR,DC arrays are provided for convenience.

```
#define NUMROWS 10
#define NUMCOLS 12

const int DR[] = {-1,0,0,1};
const int DC[] = {0,-1,1,0};

int inbounds(int row, int col);

void floodfill(int grid[][NUMCOLS], int flooded[][NUMCOLS], int row, int col) {


    if ( !inbounds(row, col)  ) return;        // Grading: 2 pts


    flooded[ row ][ col ] = 1 ;                // Grading: 1 pt

    for (int i=0; i<4; i++) {

        int nextR = row + DR[i] ;              // Grading: 2 pts

        int nextC = col + DC[i] ;              // Grading: 2 pts

        if ( grid[nextR][nextC] < grid[row][col] ) // Grading: 3 pts

            floodfill(grid, flooded, nextR, nextC);
    }

}

int inbounds(int row, int col) {
    return row >= 0 && row < NUMROWS && col >= 0 && col < NUMCOLS;
}
```

Grading Notes: Give partial credit for slots as necessary, subtract a total of 2 points if rows and columns are switched consistently (the function prototype infers that columns is the second index), don't take off for any extra checks such as seeing if flooded is 0 before doing the recursion…this turns out not to be necessary due to the acyclic structure of this specific problem.

2) (10 pts) DSN (Sorting)

The partition function in quick sort takes in an array, a low index, and a high index, which specifies a subsection of the array to partition, and returns the index where the partition element lies after performing the partition. Though there are many strategies to pick the partition element, to make grading easier, do the following: (a) use the element initially in index low to be the partition element, and (b) execute the in place partition where pairs of elements which are out of place are swapped and the partition element is swapped into its correct location at the very end right before the function returns this location. The swap function is provided for your use. **You may assume that low < high.**

```
void swap(int* ptrA, int* ptrB);

int partition(int array[], int low, int high) {

    int lowPtr = low+1, highPtr = high;

    while (lowPtr <= highPtr) {

        while (lowPtr <= high && array[lowPtr] <= array[low])
            lowPtr++;

        while (highPtr >= low && array[highPtr] > array[low])
            highPtr--;

        if (lowPtr < highPtr)
            swap(&array[lowPtr], &array[highPtr]);
    }

    swap(&array[low], &array[highPtr]);
    return highPtr;

}

void swap(int* ptrA, int* ptrB) {
    int temp = *ptrA;
    *ptrA = *ptrB;
    *ptrB = temp;
}
```

Grading: Code can be expressed in quite a few ways. Assign points to each of the following parts of the overall structure.

Outer loop set up with 2 indexes - 2 pts
Inner loop to advance low index - 2 pts
Inner loop to advance high index - 2 pts
Swap code for out of place elements - 2 pts
Last swap - 1 pt
Return - 1 pt

3) (5 pts) ALG (Bitwise Operators)

Determine the value of each of these arithmetic expressions in C. Please use the space below for your scratch work.

(i) 56 | 17 57

(ii) 47 & 83 3

(iii) 79 ^ 36 107

(iv) 13 << 3 104

(v) 187 >> 4 11

Here is the work for each part:

$56 = 111000_2$ $47 = 101111_2$ $79 = 1001111_2$
$|17 = 010001_2$ $\&83 = 1010011_2$ $\wedge 36 = 100100_2$
`------` `-------` `-------`
$111001_2 = 57_{10}$ $0000011_2 = 3_{10}$ $1101011 = 107_{10}$

$13 << 3 = 13 * 2^3 = 13 * 8 = 104$

$187 >> 4 = 187 / 2^4 = 11$

Note: Here are the last two parts using binary representation:

$13 = 1101_2$, $1101_2 << 3 = 1101000_2 = 104$

$187 = 10111011_2$, $10111011_2 >> 4 = 1011_2 = 11$

Grading: 1 pt per part, no work necessary, only answers graded.