Computer Science Foundation Exam

August 24, 2024

Section A

BASIC DATA STRUCTURES

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

SOLUTION

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 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. 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 `videogame_t` maintains information about video games that are stored in a store's inventory. As a fan of the 1980s, you would like to get a list of all the games that were produced in between the years 1980 and 1989, inclusive. Write the function below that takes in an array of type `videogame_t`, its length (`n`), and a pointer to a variable that will store the number of games produced in the 1980s. The function should return a newly dynamically allocated array storing a copy of all the information for the games that were produced in the 1980s AND set the variable pointed to by ptrNumGames to the size of the array returned by the function. This copy must be a deep copy, where individual component is copied over, including allocating memory for the copy of the name and copying the name into that new memory. **(Note: Due to the length of code, some of the function has been provided. Don't forget to allocate the appropriate space for each string in each struct!)**

```
//struct representing video game information
typedef struct {
    char * name;
    int year;
    double price;
} videogame_t;

videogame_t* getClassicGames(videogame_t * inventory,
                             int n, int* ptrNumGames) {

    videogame_t* res = malloc(n*sizeof(videogame_t));
    int nG = 0;

   for (int i=0; i<n; i++) {
       if (inventory[i].year >= 1980 && inventory[i].year < 1990) {
           res[nG].name = malloc(sizeof(char)*(1+strlen(inventory[i].name)));
           strcpy(res[nG].name, inventory[i].name);
           res[nG].year = inventory[i].year;
           res[nG].price = inventory[i].price;
           nG++;
       }
   }

    res = realloc(res, nG*sizeof(videogame_t));
    *ptrNumGames = nG;
    return res;
}
```

Grading: 1 pt for loop
3 pts if statement (take off 1 pt if -> is used)
2 pts for malloc of string (take off 1 pt if no room for '\0')
1 pt strcpy (don't give point if they use =)
1 pt copy year
1 pt copy price
1 pt update index (nG)

2) (10 pts) DSN (Linked Lists)

Complete the following user defined function that reconstructs the structure of a singly linked list. The function will take a pointer to the head of some list along with the number of nodes `n` and move the first half the nodes to the back of the list while the other half moves up to the front. **<u>For grading purposes, please write your solution iteratively, with NO recursion.</u>** The function will return the new head of the list. **<u>You may assume that the original list has an even number of elements and is not empty.</u>** The following figure shows the scenario.

Before flipHalf

After flipHalf

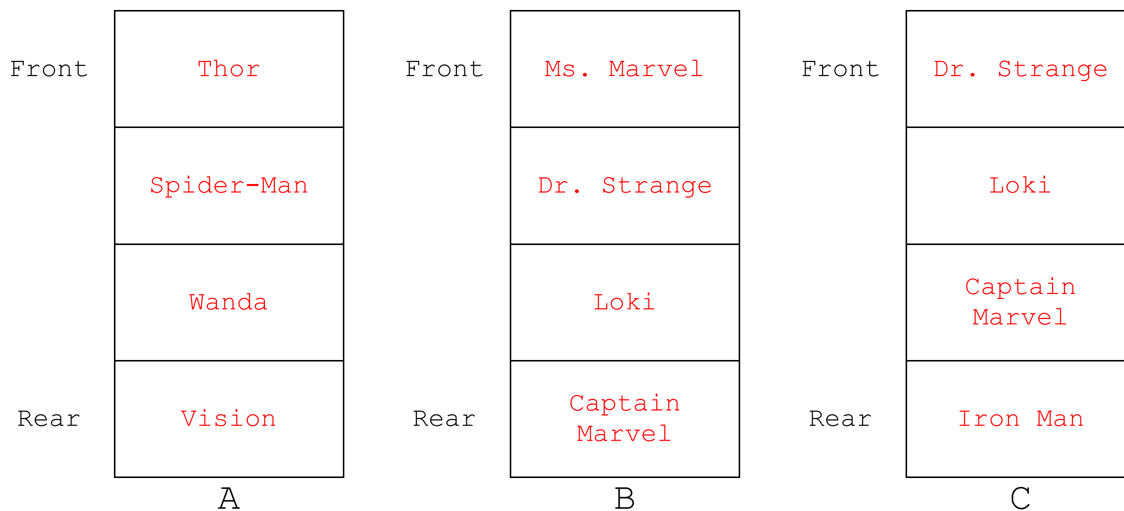
```
typedef struct node_s {
    int data;
    struct node_s* next;
} node_t;

node_t * flipHalf(node_t * head, int n) {


    node_t* end = head;
    for (int i=0; i<n/2-1; i++)
        end = end->next;

    node_t* newfront = end->next;

    end->next = NULL;

    node_t* tmp = newfront;
    while (tmp->next != NULL)
        tmp = tmp->next;

    tmp->next = head;
    return newfront;
}
```

Grading: 5 pts total – identify beginning and end of list to be returned.
1 pt – putting NULL at the end of the list to be returned.
2 pts – iterating to end of the original list.
1 pt – linking end of original list to beginning of original list.
1 pt – returning a pointer to the new front of the list.

3) (5 pts) ALG (Queues)

Consider the following C code that represents a FIFO queue that holds a list of superheroes as strings. Show the contents of the queue after each indicated point commented (A, B, and C).

```
 typedef struct node_s {
    char * hero;
    struct node_s * next;
} node_t;
typedef struct {
    node_t * front;
    node_t * back;
    int size ;
} queue_t;

void enqueue(queue_t * heroqueue, char * hero);
char * dequeue(queue_t * heroqueue);
void followQueue(queue_t * heroqueue){
      enqueue(heroqueue, "Hawkeye");
      enqueue(heroqueue, "Thor");
      enqueue(heroqueue, "Spider-Man");
      dequeue(heroqueue);
      enqueue(heroqueue, "Wanda");
      enqueue(heroqueue, "Vision"); //A
      enqueue(heroqueue, "Ms. Marvel");
      enqueue(heroqueue, "Dr. Strange");
      dequeue(heroqueue);
      dequeue(heroqueue);
      enqueue(heroqueue, "Loki");
      enqueue(heroqueue, "Captain Marvel");
      dequeue(heroqueue);
      dequeue(heroqueue); //B
      enqueue(heroqueue, "Iron Man");
      dequeue(heroqueue); //C
}
```



Note: There are exactly the correct number of boxes to be filled for each state. But, the intermediate steps may have the queue store more than 4 elements.

Grading: Stack A and Stack B are worth 2 pts. Stack C is worth 1 pt. Give partial as necessary.

Computer Science Foundation Exam

August 24, 2024

Section B

ADVANCED DATA STRUCTURES

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

SOLUTION

Last Name: ________________________________

First Name: ________________________________

UCFID: ________________________________

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

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

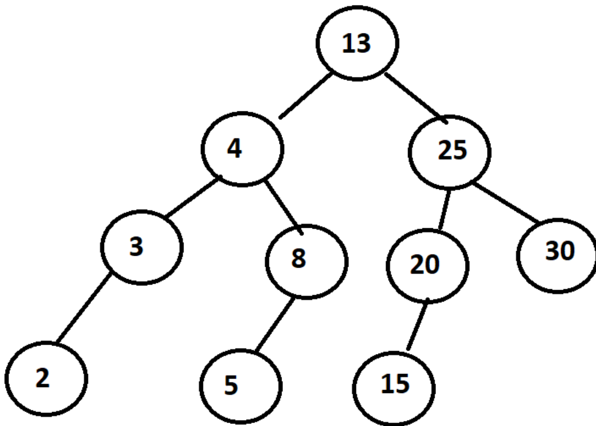
Problems will be graded based on the completeness of the solution steps and not graded based on the answer alone. Credit cannot be given unless all work is shown and is readable. Be complete, yet concise, and above all be neat. For each coding question, assume that all of the necessary includes (stdlib, stdio, math, string) for that particular question have been made.

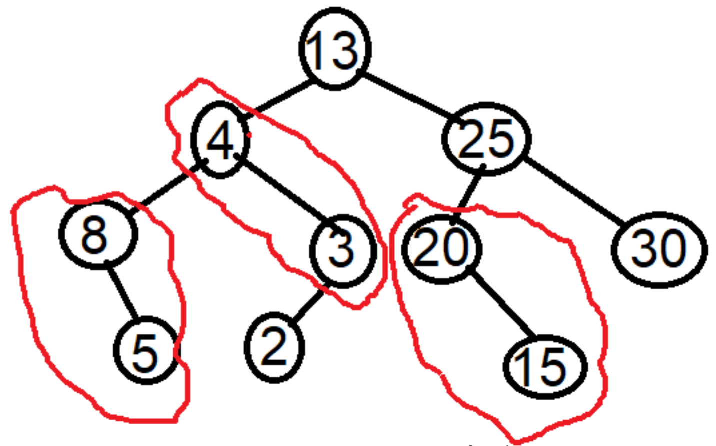
1) (10 pts) DSN (Binary Trees)

Consider the following mystery function that uses a typical tree node structure of a Binary Tree.

```
struct treenode {
     int data;
     struct treenode *left;
     struct treenode *right;
};

int mystery (struct treenode* root) {

     if(root == NULL)
         return 0;

     if(root->data %2 == 0) {
         struct treenode* temp = root->left;
         root->left = root->right;
         root->right = temp;
     }

     return 5 + mystery(root->left) + mystery(root->right);
}
```

a) Redraw (in the right side) the state of the tree below after mystery is called on its root, AND **b)** indicate the value returned by the function.





b) Return Value: ______50_____

Grading: 1 pt for exact location of each of the six nodes circled in red above, ½ pt round down for the other 4 nodes, 2 pts all or nothing for the return value.

2) (5 pts) ALG (Hash Tables)

a) (3 pts) Consider a hash table that uses the linear probing technique with the following hash function f(x) = (5x+4)%10. (The hash table is of size 10.) If we insert the values 3, 9, 2, 1, 10, and 6 into the table, in that order, show where these values would end up in the table.

Index	0	1	2	3	4	5	6	7	8	9
Value	**9**	**1**			**2**	**10**	**6**			**3**

b) (2 pts) Why is the hash function defined in part (a) a particularly bad hash function?

No matter what value is inputted into the function, there are only two possible output values: 4 and 9. We can see this because every expression of the form 5x + 4, where x is an integer ends in either 4 or 9 as the units/ones digit. This is precisely the value returned by the hash function. A good hash function provides an equal probability of each possible output value. In this case, the probability of the output equaling 0, 1, 2, 3, 5, 6, 7 or 8 is 0, thus we don't come close to providing equal probability of each of the 10 possible output values.

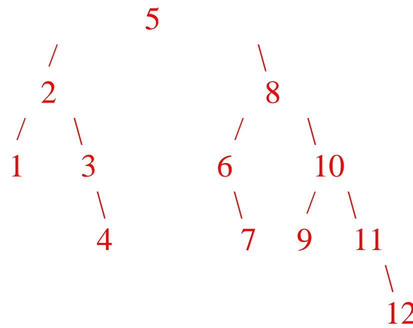
Grading: **½ for each value in the hash table, round down to an integer.**
Give full credit if answer indicates awareness that some hash values are impossible to achieve. Give partial as necessary.

3) (10 pts) ALG (AVL Trees)

There is a unique AVL tree of height 4 with 12 nodes storing the integers 1 through 12, inclusive, such that for every non-leaf node, the left subtree is strictly shorter than the right subtree.

(a) (7 pts) Draw this AVL Tree.

(b) (3 pts) Explain how you constructed the tree based on the prompt.



In order to minimize the number of nodes in an AVL tree of a fixed height, one side of the tree must have a height that is one less than the other side of the tree. Based on the directions given in the question, we must always make the left side of a node shorter than the right side. We can build up the size of the desired AVL trees of each height. Height 0 will have 1 node, Height 1 will have 2 nodes. It follows that the AVL tree with the minimal number of nodes of height 3 will have 1 (root) + 1 (left) + 2 (right) = 4 nodes. It follows that our answer must have the root node store the value 5. Using similar logic, we can deduce that the node to the left of the root node must store 2. We can then fill in the appropriate AVL trees on the left and right of the node storing 2. To fill in the right side, we note that the right node of 5 will store an AVL tree with height 3, meaning that its left subtree will have 2 nodes and its right subtree will have 4 nodes. We recursively apply the previous logic to determine that 8 is stored to the right of 5 and fill out the left and right subtrees of 8 accordingly.

Grading: 1 pt if the tree is a valid AVL tree (so must be search tree with valid heights…)
1 pt if the tree has 12 nodes storing the values 1 through 12
1 pts for having 5 at the root of the tree
1 pts total for the left subtree of 5 (give partial as needed)
3 pts for the right subtree of 5 (1 pt for 8, 1 pt for left of 8, 1 pt for right of 8)

Explanation: 3 pts total, no need to be as verbose or thorough as what's above. Give full credit for general logic in the direction of figuring out how many nodes are on the left side. Give partial credit as needed.

Computer Science Foundation Exam

August 24, 2024

Section C

ALGORITHM ANALYSIS

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

SOLUTION

Question #	Max Pts	Category	Score
1	10	ANL	
2	10	ANL	
3	5	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 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. 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)

Consider the task of sorting $\mathbf{n^2}$ integers. Using an insertion sort, this task would take $O(\mathbf{n^4})$ time. Using a single heap sort, this task would take $O(\mathbf{n^2 lg\ n})$. Consider this hybrid approach and, **with proof, determine its worst case run time, in terms of n.** Assume efficient implementations of each of the heap and linked list operations described. Leave your answer in Big-Oh notation.

1. Separate the $\mathbf{n^2}$ integers into **n** groups of **n** integers each.

2. Create heaps out of each of the **n** groups of integers.

3. Call delete min on each of the **n** heaps, storing these **n** deleted values in a linked list, also storing which heap each value came from.

4. Repeat the following $\mathbf{n^2}$ times:

    a. Loop through the linked list, locating the minimum integer in it, noting which heap it was from. Name the integer **x** and the heap **H**.

    b. Place **x** next in the sorted list and delete it from the linked list.

    c. If **H** isn't empty, delete the minimum item from **H** and add it to the end of the linked list, also storing that the value came from heap **H**.

The first step takes $O(n^2)$ time as each number if processed once.

The run-time of make heap on n integers is $O(n)$. Since we do this n times, the total run-time of step 2 is $O(n^2)$.

Each delete min call on a heap takes $O(\lg n)$ time because each of the heaps from which we delete items will have no more than n items. Ultimately, each item from each heap gets deleted, so the total run time of all the delete mins on the heap is $O(n^2 \lg n)$ time. This encompasses the total run time of steps 3 and 4c.

In step 4, the loop in part (a) takes $O(n)$ time and we repeat it n^2 time, meaning that we spend a total of $O(n^3)$ time on step 4a across all of its iterations.

Step 4b takes $O(1)$ time each time it's executed for a total of $O(n^2)$ time.

If we add up all of these run-times we get:

$O(n^2) + O(n^2) + O(n^2 \lg n) + O(n^3) + O(n^2) = \mathbf{O(n^3)}$

An alternate way to do the analysis is to state that step 3 takes $O(n \lg n)$ time, followed by a single iteration of step 4 taking $O(n) + O(1) + O(\lg n) = O(n)$ time, then noting that since this gets repeated n^2 times, the total run time of step 4 is $O(n^3)$,

Grading: 2 pts for the correct answer
2 pts for explaining why step 2 takes either $O(n^2)$ or $O(n^2 \lg n)$ time
3 pts for multiplying n^2 times the run time of steps 4a, 4b and 4c.
3 pts for the run time of steps 4a, 4b and 4c as well as step 3 with explanation

Note: The last 8 points are based on the level of explanation, give partial as you see fit.

2) (10 pts) ANL (Algorithm Analysis)

A brute force algorithm which processes all permutations of n routines runs in $O(n \times (n!))$ time. On a particular computer, executing the algorithm for **n = 9** takes 180 milliseconds. How many seconds is the algorithm expected to take on an input of size **n = 11**, run on the same computer?

Let $T(n) = cn(n!)$ be the execution time for the algorithm run on an input of size n. Using the given information we have:

$$T(9) = c9(9!) = 180\ ms \rightarrow c = \frac{180\ ms}{9(9!)} = \frac{20\ ms}{9!}$$

Let's now solve for T(11):

$$\begin{aligned} T(11) = c(11)(11!) &= \frac{20ms}{9!} \times 11 \times 11! \\ &= 20ms \times \frac{11(11!)}{9!} \\ &= 20ms \times \frac{(11)9!(10)(11)}{9!} \\ &= 20ms \times 11 \times 10 \times 11 \\ &= 20ms \times 1210 \\ &= 24200ms \\ &= \mathbf{24.2\ seconds} \end{aligned}$$

Grading: 1 pt set up equation for c
3 pts solve for c, no simplification necessary
1 pt set up expression for T(11)
1 pt substitute value of c and 11
3 pts for algebraic simplification
1 pt converting to seconds

3) (5 pts) ANL (Summations)

Determine a closed form solution to the following summation in terms of **n**. Simplify your answer to standard polynomial form (not factored).

$$\sum_{i=1}^{3n}(2i+5)$$

$$\sum_{i=1}^{3n}(2i+5) = \sum_{i=1}^{3n}2i + \sum_{i=1}^{3n}5$$

$$= \frac{2(3n)(3n+1)}{2} + 5(3n)$$

$$= 3n(3n+1) + 15n$$

$$= 9n^2 + 3n + 15n$$

$$= \mathbf{9n^2 + 18n}$$

Grading: 1 pt split sum, 2 pts apply sum of 2i, 1 pt apply sum of 5, 1 pt simplify to right form

Computer Science Foundation Exam

August 24, 2024

Section D

ALGORITHMS

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

SOLUTION

Question #	Max Pts	Category	Score
1	5	DSN	
2	10	DSN	
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 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. For each coding question, assume that all of the necessary includes (stdlib, stdio, math, string) for that particular question have been made.

1) (5 pts) DSN (Recursion)

Write a recursive function that given an array, the length of the array, and a range of indices finds the number of even integers in an array in the specified index range. **Solutions that use a loop will receive 0 points**. (For example, if the arr stored [3, 5, 8, 6, 7, 9, 4, 11, 8], the function call evens(arr, 9, 2, 6) should return 3 because the contents of index 2, 3 and 6 are all even numbers. In particular, arr[2] = 8, arr[3] = 6 and arr[6] = 4.)

```
int evens(int * arr, int len, int lowInd, int highInd) {

    if (lowInd > highInd) return 0;

    return (arr[lowInd]%2 == 0) + evens(arr, len, lowInd+1, highInd);

}
```

Grading: Base case – 2 pts (could be lowInd == highInd as well)
Recursive call – 2 pts (multiple ways to make this)
Return and adding the right amount – 1 pt
Note: The grading criteria is intentionally generous.

2) (10 pts) DSN (Sorting)

For this problem, fill in the blank to finish the stable (elements with the same values are kept in their original order) quicksort on a linked list. We are using the head of the linked list as a pivot. You can assume that the following linked list functions have all been implemented and take O(1) operations.

Note: Each blank is worth one point and involves either making calls or filling in parameters to the functions whose prototypes and descriptions are given below.

```
typedef struct Node {
    int value;
    struct Node * next;
} Node;

typedef struct List {
    Node* front;
    Node* back;
} List;

void addToTail(List * list, Node * node); // Add to tail

// Returns a list that is the combination of 2 given lists.
List * merge(List * front, List * back);

Node* getAndRemoveHead(List * list); // Removes and returns the head
List* createEmptyList(); // Returns dynamically allocated empty List
int isEmpty(List * list); // Returns 1 if empty and 0 otherwise
void deleteList(List * list); // Cleans up any leftover dynamic
memory

// Sort code on next page
```

```
List * sort(List * lst) {

   if (__isEmpty__(lst)) return lst;

   Node * pivot = getAndRemoveHead(lst);
   List * first = __createEmptyList__();
   List * last = __createEmptyList__();
   List * middle = createEmptyList();

   addToTail(middle, pivot);
   while (!isEmpty(lst)) {
      Node * cur = __getAndRemoveHead(lst);
      if (cur->value < pivot->value)
         addToTail(__first__, cur);
      else if (cur->value == pivot->value)
         addToTail(__middle__, cur);
      else
         addToTail(__last__, cur);
   }

   first = sort(__first__);
   last = sort(__last__);

   first = merge(first, middle);
   first = merge(first, last);
   free(middle);
   free(last);
   free(lst);

   return __first__;
}
```

(+1 pt) per blank

3) (10 pts) DSN (Bitwise Operators)

A lights out puzzle is a classic puzzle played on a 2D grid that involves turning the cells of the grid off. The cells can be toggled by pressing a particular cell. Doing so will invert the cell pressed and the 4 adjacent cells that are the immediate neighbor (joined by a cell line). Your program stores a "lights out" puzzle represented using an array of integers, where each integer in the array represents a row of the puzzle. The bits when on represent that the cell is on, and when off represent that the cell is off. If in some row the first, sixth, and seventh cell is on, then the value at the spot in the array would be $97 = 2^0 + 2^5 + 2^6$. If we were to press the button in column 1 of this row, then it would turn spot 0 off, spot 1 on, and spot 2 on, changing the bit mask on that row to $2^1 + 2^2 + 2^5 + 2^6 = 102$. In addition, if the row above existed one bit would be changed there and if the row below existed, one bit would be changed there.

You will create a function that will emulate pressing a particular cell. You will be given the row and column (0-based index) of the cell toggled along with the grid and its dimensions. Modify the grid based on the described interaction. Be careful of indexing out of bounds. (Hint: There are 5 bits total that will potentially flip. One of those bits will always flip while the other four bits, the neighboring bits, will only flip if they are within the bounds of the grid.)

```
void cellPress(int row, int col, int height, int width, int * grid) {

    grid[row] ^= (1<<col);

    if (col)
        grid[row] ^= (1<<(col-1));

    if (row)
        grid[row-1] ^= (1<<(col));

    if (row < height - 1)
        grid[row+1] ^= (1<<(col));

    if (col < width - 1)
        grid[row] ^= (1<<(col+1));
}
```

Grading: 2 pts for each of the five sections of code. For the ones with the if statements, 1 point for the if and 1 pt for the toggle.

Note: Due to some confusion during the exam, give full credit for a valid 2D array solution where grid is two dimensional, the appropriate out of bounds checks are made and the corresponding XORs (all with 1 because presumably 0 or 1 would be stored in each cell of a 2D array) occur.