# ECE264 Fall 2019 Final Exam

# 8-10 AM, December 12
# Sections 1 and 2
# (Dr. Lu's sections)

Please keep the answer sheet clean. Do not use the answer sheet as your scratch space. The answer sheet should contain **only** the answers.

Do not write anything that is not the answers.
Otherwise, you may lose points.

Please use **DARK** ink. If your pen is too light, your answer may not be graded.
Please write at the center of each box.

# Enjoy the Semester Break

The values below are decimal.

| Value | Character | Value | Character |
|---|---|---|---|
| 65 | A | 97 | a |
| 66 | B | 98 | b |
| 67 | C | 99 | c |
| 68 | D | 100 | d |
| 69 | E | 101 | e |
| 70 | F | 102 | f |
| 71 | G | 103 | g |
| 72 | H | 104 | h |
| 73 | I | 105 | i |
| 74 | J | 106 | j |
| 75 | K | 107 | k |
| 76 | L | 108 | l |
| 77 | M | 109 | m |
| 78 | N | 110 | n |
| 79 | O | 111 | o |
| 80 | P | 112 | p |
| 81 | Q | 113 | q |
| 82 | R | 114 | r |
| 83 | S | 115 | s |
| 84 | T | 116 | t |
| 85 | U | 117 | u |
| 86 | V | 118 | v |
| 87 | W | 119 | w |
| 88 | X | 120 | x |
| 89 | Y | 121 | y |
| 90 | Z | 122 | z |

# 1 Huffman Compression

## 1.1 Codes

Consider the following Huffman tree.

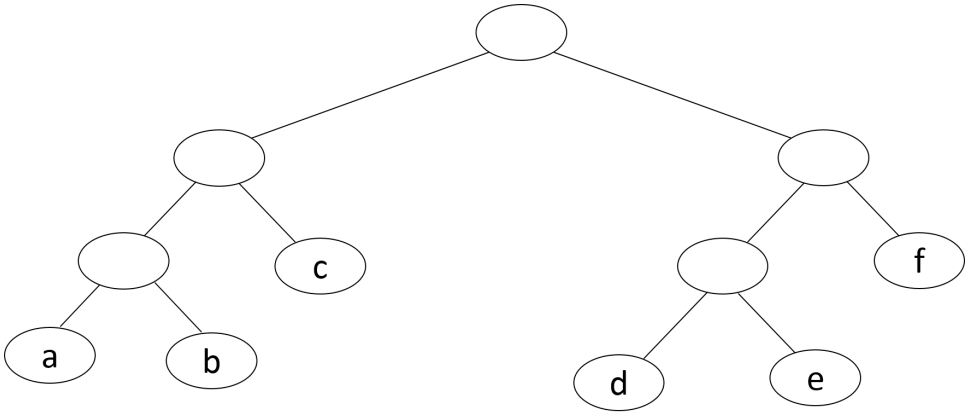



Figure 1: Huffman Tree.

What is the code for `e`?

## 1.2 Properties and Codes of Huffman Tree

Choose **every** correct statement.
If a statement is correct and you do not choose it, you lose points for that statement.
If a statement is incorrect and you choose it, you lose points for that statement.

1. The code of every letter has the same number of bits.
2. If one letter appears more frequently than another letter, the code for the first letter must always be shorter than the code for the second letter.
3. In Figure 1, the frequency of `c` must be the same as the frequency of `f`.
4. The shape of a Huffman tree depends on the content to be compressed.
5. The code for `a` is 110.
6. The code for `b` is 010.
7. The code for `c` is 011.

## 1.3 Express Huffman Tree using Post-Order

If a Huffman tree is expressed using post-order as

```
1c 1d 1a 0 1b 0 1e 1f 1g 0 0 0 0 0
```

Space is added for clarity. The space does not represent any information.
What is the code for `b`?

## 1.4 Bit Expression of Huffman Tree

Continue from the previous question using `1c 1d 1a 0 1b 0 1e 1f 1g 0 0 0 0 0` as the Huffman Tree. Write the **second** byte in hexadecimal (i.e., starting with `0X`). You do not need to write `0X`. Please refer to the ASCII table earlier in this exam. Each letter uses 8 bits.

## 1.5 Huffman Compression Data

(This question is not related to 1.3 or 1.4.)
This table shows the codes of letters:

| | |
|---|---|
| `a` | 0 0 |
| `b` | 0 1 |
| `c` | 1 |

If the beginning of the information is

a b c a b c c a b a b c ...

here ... means additional letters. Space is added for clarity. The space does not represent any information.
Write down the **second** byte in hexadecimal (i.e., starting with `0X`). You do not need to write `0X`.
**Answer:**

1. 101
2. 4
3. 101
4. 0XD9 or 217, it is ok not to include 0X

   Explanation: c is 99 = 0110 0011, d is 100 = 0110 0100.

   The tree is `1 0110 0011 1 0110 0100` = `1011 0001 1101 1001 00` = `0X--D9`.
5. 0XE2 or 226

   Explanation

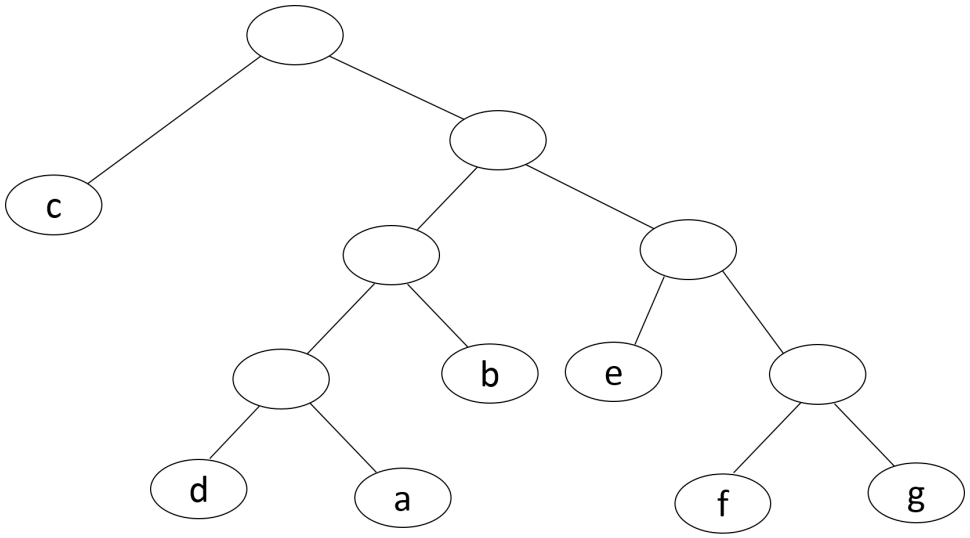



Figure 2: Huffman Tree for `1c 1d 1a 0 1b 0 1e 1f 1g 0 0 0 0 0`.

```
a  b  c  a  b  c  c  a  b  a  b  c ...
00 01 1  00 01 1  1  00 01 00 01 1
```

Regroup:

```
0001 1000 1110 0010
```

# 2 Build Huffman Tree

The following table shows how many times each letter occurs in a file.

| Letter | Occurrences |
|---|---|
| a | 20 |
| b | 6 |
| c | 5 |
| d | 1 |
| e | 24 |
| f | 13 |
| g | 2 |

Create the Huffman tree.

## 2.1 Sibling of f

Which letter is the sibling of f (i.e., sharing the same parent node)? If f's silbing is not a letter, please write -.

## 2.2 Code of e?

What is the code of e?
For your reference, the code of a is 10.

## 2.3 Code of b?

What is the code of b?
For your reference, the code of c is 0111.

## 2.4 Code of g

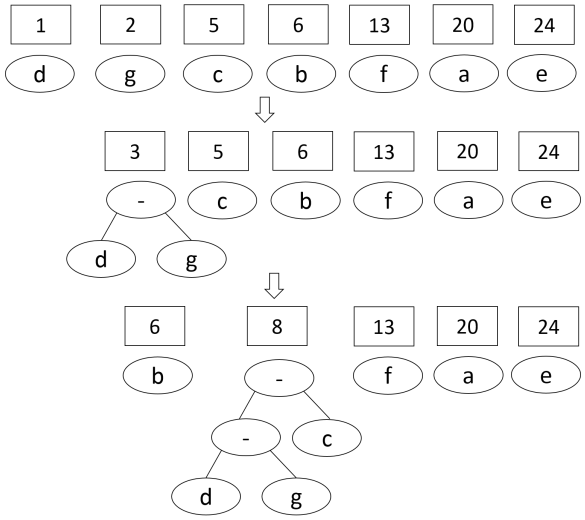
What is the code of g?

## 2.5 Letter with 2-bit codes

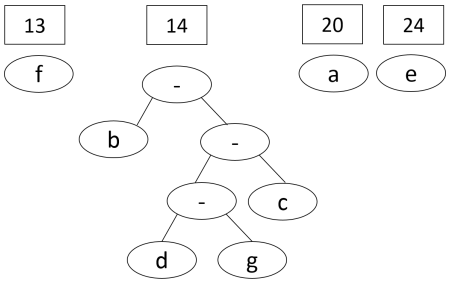
How many letters have 2-bit codes?

**Answer:**

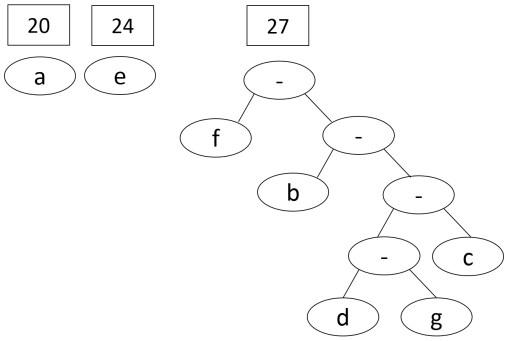
1. -
2. 11
3. 010
4. 01101 or 01100
5. 3: f, a, e


1 2 5 6 13 20 24
d g c b f a e
3 5 6 13 20 24
- c b f a e
d g
6 8 13 20 24
b - f a e
- c
d g

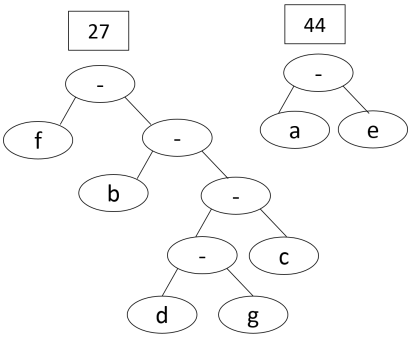
(a)


13 14 20 24
f - a e
b -
- c
d g

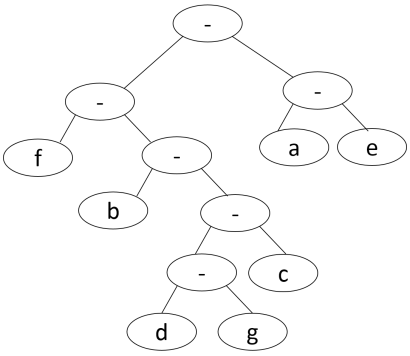
(b)


20 24 27
a e -
f -
b -
- c
d g

(c)


27 44
- -
f - a e
b -
- c
d g

(d)


-
- -
f - a e
b -
- c
d g

(e)

# 3 Maze

Consider HW 17 Maze.

```
1  #ifndef MAZE_H
2  #define MAZE_H
3
4  typedef struct
5  {
6    int * * cells;
7    // two dimensional array to store each cell:
8    // -1: brick
9    //  0: starting point
10   // a large number (numrow * numcol + 1): cell not yet visited
11   int numcalls; // how many times the move function is called?
12   // size of the maze
13   int numrow;
14   int numcol;
15   // starting location
16   int startrow;
17   int startcol;
18
19   // additional attributes if necessary
20   // ...
21 } Maze;
22
23 #endif
```

The function `findDistance` finds the shortest distance from the starting point by calling `move`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "maze.h"
4
5  enum {ORIGIN, EAST, SOUTH, WEST, NORTH};
6
7  void printMaze(Maze * maz)
8  {
9    if (maz == NULL)
10     {
11       return;
12     }
13   int indrow;
14   int indcol;
```

```
15    for (indrow = 0; indrow < (maz -> numrow); indrow ++)
16      {
17          for (indcol = 0; indcol < (maz -> numcol); indcol ++)
18            {
19              printf("%4d ", (maz -> cells)[indrow][indcol]);
20            }
21          printf("\n");
22      }
23  }
24
25  static int canMove(Maze * maz, int row, int col, int dir)
26  {
27    /* (row, col) is the current location */
28    switch (dir)
29      {
30      case NORTH:
31        row --;
32        break;
33      case SOUTH:
34        row ++;
35        break;
36      case WEST:
37        col --;
38        break;
39      case EAST:
40        col ++;
41        break;
42      }
43    // is the index valid?
44    if ((row < 0) || (row >= (maz -> numrow))) { return -1; }
45    if ((col < 0) || (col >= (maz -> numcol))) { return -1; }
46
47    return (maz->cells)[row][col];
48  }
49
50  static void move(Maze * maz, int row, int col, int distance)
51  {
52    int dest;
53    (maz -> numcalls) ++;
54    (maz -> cells)[row][col] = distance;
55    // move if the a neighbor's distance is larger than the
56    // current distance + 1
```

```
57    dest = canMove(maz, row, col, EAST);
58    if (dest > (distance + 1))
59      {
60        move(maz, row, col + 1, distance + 1);
61      }
62    dest = canMove(maz, row, col, WEST);
63    if (dest > (distance + 1))
64      {
65        move(maz, row, col - 1, distance + 1);
66      }
67    dest = canMove(maz, row, col, NORTH);
68    if (dest > (distance + 1))
69      {
70        move(maz, row - 1, col, distance + 1);
71      }
72    dest = canMove(maz, row, col, SOUTH);
73    if (dest > (distance + 1))
74      {
75        move(maz, row + 1, col, distance + 1);
76      }
77  }
78
79  void findDistance(Maze * maz)
80  {
81    if (maz == NULL)
82      {
83        return;
84      }
85    printMaze(maz);
86    maz -> numcalls = 0;
87    move(maz, maz -> startrow, maz -> startcol, 0);
88    printf("move is called %d times\n", maz -> numcalls);
89    printMaze(maz);
90  }
```

The `Maze` structure has an integer, `numcalls`, to count how many times the `move` function is called. Please be aware that the `move` function is called the very first time from the starting point and the distance of this cell is zero.

For this set of questions, the "**correctness**" of the program is determined by whether the output of `printMaze` called by `findDistance` at line 89 is correct or not for any valid input.

## 3.1 Simple Maze

Consider this maze (the line numbers are added for your reference and are not parts of the maze).

```
1 bb bb
2 bb  b
3 b b b
4 b b b
5 bbbsb
6 bbbbb
```

This is the output of the program:

```
  -1   -1   43   -1   -1
  -1   -1   43   43   -1
  -1   43   -1   43   -1
  -1   43   -1   43   -1
  -1   -1   -1    0   -1
  -1   -1   -1   -1   -1
move is called ???? times
  -1   -1    5   -1   -1
  -1   -1    4    3   -1
  -1   43   -1    2   -1
  -1   43   -1    1   -1
  -1   -1   -1    0   -1
  -1   -1   -1   -1   -1
```

How many times has `move` been called (i.e., what is `???`)?

## 3.2 Complex Maze

Next, consider a complex maze:

```
1 b bbbbb
2 b b   b
3 b   bsb
4 b b b b
5 b  b  b
6 bbbbbbb
```

This is the output of the program:

```
  -1   49   -1   -1   -1   -1   -1
  -1   49   -1   49   49   49   -1
  -1   49   49   49   -1    0   -1
```

```
  -1   49   -1   49   -1   49   -1
  -1   49   49   -1   49   49   -1
  -1   -1   -1   -1   -1   -1   -1
move is called ??? times
  -1    8   -1   -1   -1   -1   -1
  -1    7   -1    3    2    1   -1
  -1    6    5    4   -1    0   -1
  -1    7   -1    5   -1    1   -1
  -1    8    9   -1    3    2   -1
  -1   -1   -1   -1   -1   -1   -1
```

How many times has `move` been called (i.e., what is `???`)?

## 3.3 Properties of `maze.c`

Choose **one** (only one) correct statement.

1. The `move` function tries to move in the order of `EAST`, `WEST`, `NORTH`, and `SOUTH`. If the order is changed, the `move` function may be called different numbers of times, even for the same maze. In order words, the program's output

   ```
   move is called ???  times
   ```

   **may** have different values at `???` if the order is changed.

2. The `move` function tries to move in the order of `EAST`, `WEST`, `NORTH`, and `SOUTH`. If the order is changed, the shortest distances (the final results) of some cells may be different, even for the same maze.

3. The program sets the distance for bricks to `-1`. The program will still function correctly if the distance for bricks are set to a large positive number, as long as the number is larger than `2 * number of rows * number of column + 1`.

4. The program sets the distance for bricks to `-1`. If the distance for bricks are set to `0`, the program will **not** function correctly

5. If a maze has a cell that can be reached by multiple paths, this recursive function `move` will **not** function correctly. It is possible that this recursive function keeps calling itself indefintely and runs out of the stack memory.

### 3.4 True / False: Where to set distance

Write T (true) or F (false) whether the following statement is correct.

*"Currently,*

```
(maz -> cells)[row][col] = distance;
```

*is before the four* `if` *blocks.*
*The program will still function correctly if*

```
(maz -> cells)[row][col] = distance;
```

*is moved after the four* `if` *blocks."*

Please write only T or F. **DO not write anything else.** You will receive **no** point if you write True, true, False, or false.

### 3.5 True / False: Multiple Exits

Each of the two mazes shown above has only one exit at the top.
Write T (true) or F (false) whether the following statement is correct.

*"The program will* **not** *function correctly if a maze has multiple exits at the top.*

Please write only T or F. **DO not write anything else.** You will receive **no** point if you write True, true, False, or false.
**Answer:**

1. 6
2. 16
3. 1
4. T
5. F

# 4 Memory Management

Consider HW 18 Memory Management.
Suppose `createMemory`'s second argument (i.e., `size`) is 16 to create 16 memory blocks.
This is the state of the memory before calling `allocateMemory` or `freeMemory`.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |

For your convenience, the following tables are available to write down the process. Please remember that you **must** write the answers on the answer sheet.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |

## 4.1 allocateMemory and freeMemory

Consider the following testing program:

```
1  int main(int argc, char * * argv)
2  {
3    int allocsize[4] = {2, 4, 3, 5}; // allocate four times
4    int startmem[4] = {-1}; // starting addresses
5    int memsize = 16;
6    Memory * mem;
7    createMemory(& mem, memsize);
8    startmem[0] = allocateMemory(mem, allocsize[0]);
9    startmem[1] = allocateMemory(mem, allocsize[1]);
10   startmem[2] = allocateMemory(mem, allocsize[2]);
11   freeMemory(mem, startmem[1]);
12   saveOccupancy(mem, "result1");
13   startmem[3] = allocateMemory(mem, allocsize[3]);
```

```
14    saveOccupancy(mem, "result2");
15    destroyMemory(mem);
16    return EXIT_SUCCESS;
17  }
```

After the program finishes,

A. What is the second bit of the second byte in `result1`?

B. What is the third bit of the second byte in `result2`?

C. What is the value stored in `startmem[1]`?

## 4.2 Allocation Fail (Answer D)

Which of the following sequence will received -1 when calling `allocateMemory`?
Choose **every** correct statement if **at least one** `allocateMemory` returns -1.
If a statement is correct and you do not choose it, you lose points for that statement.
If a statement is incorrect and you choose it, you lose points for that statement. Use `startmem` to store the starting addresses.

```
int startmem[100] = {-1};
```

1.
```
startmem[0] = allocateMemory(mem, 4);
startmem[1] = allocateMemory(mem, 4);
startmem[2] = allocateMemory(mem, 4);
startmem[3] = allocateMemory(mem, 4);
startmem[4] = allocateMemory(mem, 4);
```

2.
```
startmem[0] = allocateMemory(mem, 4);
startmem[1] = allocateMemory(mem, 4);
startmem[2] = allocateMemory(mem, 4);
startmem[3] = allocateMemory(mem, 4);
freeMemory(mem, startmem[1]);
freeMemory(mem, startmem[2]);
freeMemory(mem, startmem[0]);
startmem[1] = allocateMemory(mem, 4);
```

3.
```
startmem[0] = allocateMemory(mem, 4);
startmem[1] = allocateMemory(mem, 4);
startmem[2] = allocateMemory(mem, 4);
startmem[3] = allocateMemory(mem, 4);
freeMemory(mem, startmem[1]);
freeMemory(mem, startmem[3]);
startmem[1] = allocateMemory(mem, 5);
```

4.
```
startmem[0] = allocateMemory(mem, 4);
startmem[1] = allocateMemory(mem, 3);
startmem[2] = allocateMemory(mem, 5);
startmem[3] = allocateMemory(mem, 4);
freeMemory(mem, startmem[0]);
freeMemory(mem, startmem[2]);
startmem[0] = allocateMemory(mem, 6);
```

5.
```
startmem[0] = allocateMemory(mem, 3);
startmem[1] = allocateMemory(mem, 4);
startmem[2] = allocateMemory(mem, 3);
startmem[3] = allocateMemory(mem, 5);
freeMemory(mem, startmem[0]);
freeMemory(mem, startmem[3]);
startmem[0] = allocateMemory(mem, 4);
```

## 4.3 Allocated Address (Answer E)

Consider the following testing program:

```
1  int main(int argc, char * * argv)
2  {
3    int startmem[10] = {-1}; // starting addresses
4    int memsize = 16;
5    Memory * mem;
6    createMemory(& mem, memsize);
7    startmem[0] = allocateMemory(mem, 2);
8    startmem[1] = allocateMemory(mem, 3);
9    startmem[2] = allocateMemory(mem, 4);
10   startmem[3] = allocateMemory(mem, 5);
11   freeMemory(mem, startmem[1]);
12   freeMemory(mem, startmem[3]);
13   startmem[4] = allocateMemory(mem, 4);
14   startmem[5] = allocateMemory(mem, 2);
15   destroyMemory(mem);
16   return EXIT_SUCCESS;
17 }
```

What is the value of `startmem[5]` before the program calls `return EXIT_SUCCESS`?
**Answer:**

1. 0: result1 is 11000011 10000000
2. 1: result2 is 11000011 11111100

3. 2

4. 1, 3, 4

5. 2