ECE368 Exam 2 Spring 2018

Thursday, April 12, 2018, 6:30-7:30pm
LILY 1105

Sign the following statement AND write down your name. If the statement is not signed, the exam will not be graded and it will not be returned.

I pledge on my honor that I have neither given nor received any unauthorized assistance on this exam. If I fail to honor this pledge, I will receive a failing grade for this course and be subject to disciplinary action.

Signature:

First Name (Given Name):

Last Name (Family Name):

This is an *open-book, open-notes* exam. Electronic devices are not allowed.

This exam consists of 8 pages; it is *your responsibility* to make sure that you turn in a complete copy of the exam.

If you score 50% or more for a question, you are considered to have met all learning objectives associated with that question. Learning objectives 1–5 are covered in this exam.

Be *concise*. When you are asked of the time complexity of an algorithm, it is *not* necessary to do a line-by-line analysis of the algorithm. *A general explanation would suffice*.

Assume that all necessary "`.h`" files are included and all `malloc` function calls are successful.

If you are not clear about what a question is asking, you can explain what you are answering (e.g., "I think this question is asking for ...") or you can state assumptions that you are making (e.g., "I assume that the entry -1 is equivalent to NULL").

Your Purdue ID should be visible for us to verify your identity.

If you finish the test before 7:25pm, you may turn in the test and leave. **After 7:25pm, you have to stay until we release the entire class. Stop writing at 7:30pm. If you continue to write, that is considered cheating.**

When we collect the copies of test, you are to remain in your seat in an orderly manner until we have collected and counted all copies.

Alphabetical order: a b c d e f g h i j k l m n o p q r s t u v w x y z

DO NOT BEGIN UNTIL INSTRUCTED TO DO SO ...

1 Quick Sort (20 points) (Learning Objectives 2 and 3)

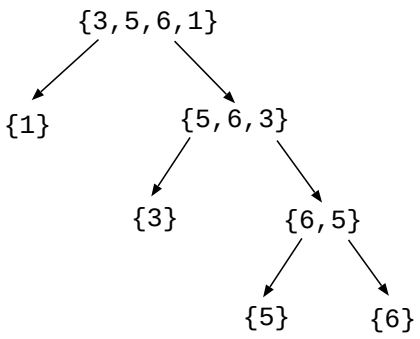
Consider the following quick sort algorithm covered in **HW07 Question 1 (not lecture notes)**.

```
1  int partition(int *array, int lb, int ub)
2  {
3     int pivot = array[lb];
4     int down = lb - 1;
5     int up = ub + 1;
6     while (true) {
7        do {
8           up--;
9        } while (array[up] > pivot);
10       do {
11          down++;
12       } while (array[down] < pivot);
13       if (down < up) {
14          int tmp = array[down];
15          array[down] = array[up];
16          array[up] = tmp;
17       } else {
18          return up;
19       }
20    }
21 }
22
23 void quick_sort(int *array, int lb, int ub)
24 {
25    if (lb >= ub) {
26       return;
27    }
28    int partition_idx = partition(array, lb, ub);
29    quick_sort(array, lb, partition_idx);
30    quick_sort(array, partition_idx+1, ub);
31 }
```

(a) (6 points) The following statements are in the `main` function.

```
int array[] = {3, 1, 5, 4, 4, 2, 3, 5};
quick_sort(array, 0, sizeof(array)/sizeof(array[0]) - 1);
```

Draw a computation tree that shows the recursive calls of the function `quick_sort` on `array`. As an example, the figure in the next page shows the computation tree of performing `quick_sort` on {`3, 5, 6, 1`}, where each node is represented by the array passed to the function.



(b) (8 points) As covered in the lecture, the replacement of an appropriate tail recursion with iterations (**lines 40 and 44**) as follows can reduce the call stack usage:

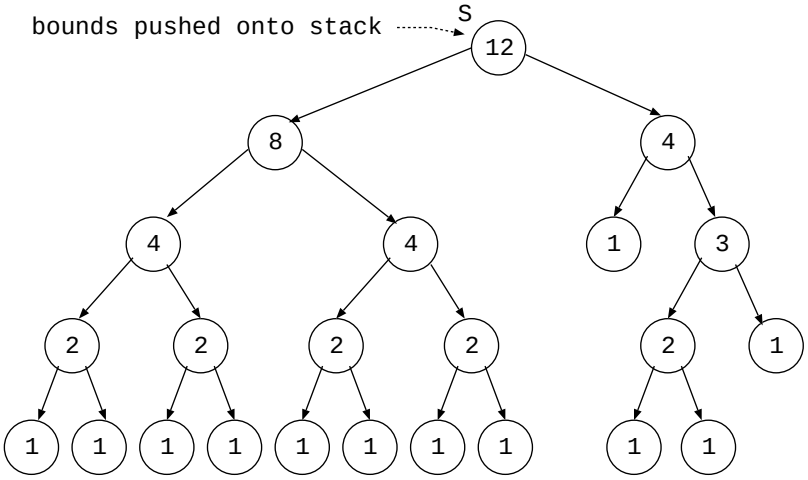
```
32 void quick_sort_tail_removed(int *array, int lb, int ub)
33 {
34    while (ub > lb) {
35       int partition_idx = partition(array, lb, ub);
36       // Does left partition have more elements than right?
37       if (ub - partition_idx < partition_idx - lb + 1) {
38          // left is tail recursion, replaced by iterations
39          quick_sort_tail_removed(array, partition_idx+1, ub);
40          ub = partition_idx;   // left
41       } else {   // left <= right
42          // right is tail recursion, replaced by iterations
43          quick_sort_tail_removed(array, lb, partition_idx);
44          lb = partition_idx+1; // right
45       }
46    }
47 }
```

Going one step further, Prof. Koh removes the remaining recursion by maintaining a stack in the pseudo-code shown in the next page. Here, we assume that you can check whether a stack is empty (`is_empty`), push a pair of bounds onto a stack (`push`), and pop from a stack a pair of bounds (`pop`).

First, we push the bounds of the array onto the stack (**line 50**). While the stack is not empty, we pop a pair of bounds (`lb`, `ub`) from the stack and sort the corresponding array. After partitioning, instead of passing lower and upper bounds into a recursive function (**lines 39 or 43** in `quick_sort_tail_removed`), we push the bounds onto a stack (**lines 59 or 64**, respectively).

```
48 void quick_sort_no_rec(int *array, int lb, int ub)
49 {
50    push(stack, (lb, ub));
51    while (is_empty(stack) == false) {  // while stack not empty
52       (lb, ub) = pop(stack);
53       while (ub > lb) {
54          int partition_idx = partition(array, lb, ub);
55          // Does left partition have more elements than right?
56          if (ub - partition_idx < partition_idx - lb + 1) {
57             // left is tail recursion, replaced by iterations
58             // use stack to remember right recursion
59             push(stack, (partition_idx+1, ub));
60             ub = partition_idx;    // left
61          } else {   // left <= right
62             // right is tail recursion, replaced by iterations
63             // use stack to remember left recursion
64             push(bounds_stack, (lb, partition_idx));
65             lb = partition_idx+1;  // right
66          }
67       }
68    }
69 }
```

The following is a computation tree for the version of quick sort in **Question 1(a)** that makes two recursive function calls, where the number in each node corresponds to the size of the array to be sorted by the recursive function call. Prof. Koh's implementation changes some function calls into iterations and pushes the bounds of some function calls onto the stack. Beside each node in the computation tree, write down upper-case "I" if the function call has been changed into iterations or upper-case "S" if its bounds have been pushed onto the stack. The root node of the computation tree is labeled as "S" because the bounds have been pushed onto the stack.

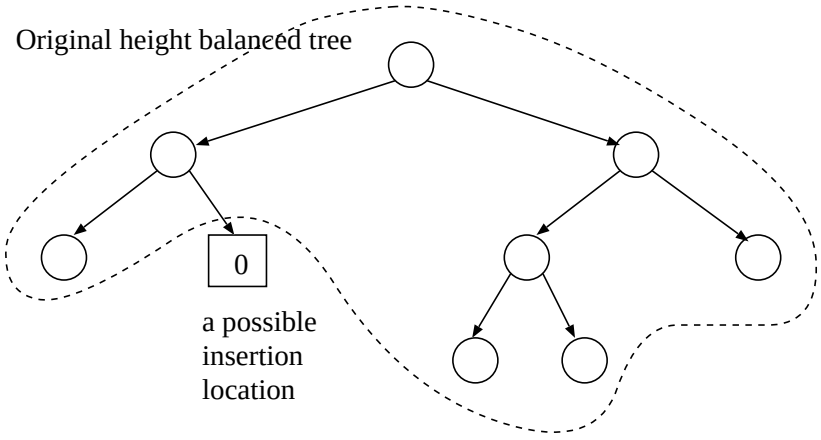


(c) (6 points) For Prof. Koh's implementation in **Question 1(b)**, what is the worst-case space complexity in terms of n, where n is the size of the array to be sorted? Justify your answer.

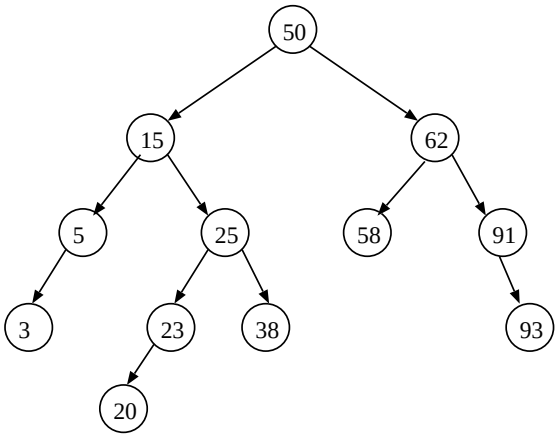
If the worst-case space complexity is not $O(\log n)$, make necessary changes to Prof. Koh's implementation in **Question 1(b)** such that the implementation has $O(\log n)$ worst-case space complexity. Your answer can be in the form "Change line 48: ..." or "Insert between lines 48 and 49: ...". You may also make changes in Page 4 directly. If the worst-case space complexity is indeed $O(\log n)$, leave this part blank.

2 Height-Balanced Trees (20 points) (Learning Objective 1)

(a) (12 points) Consider the following height-balanced binary search tree (with tree nodes depicted as circles). We also show a rectangle that is a possible location to insert a key into the binary search tree before performing necessary rotation(s). The number in the rectangle is the number of rotations that need to be performed when the new key is inserted at this location. Identify all other possible locations (before performing any necessary rotations) when a new key is inserted into the original binary search tree. Use a rectangle to indicate a possible insertion location and indicate in each rectangle the number of rotations that you have to perform to maintain height balancedness.



(b) (8 points) Consider the following binary search tree that is height-balanced. Show the final height-balanced tree after the deletion of the key 50 from the original tree. When you delete a node that has two children, replace that node with its **immediate successor**. **Identify the replacement key** and **specify the rotations performed** using arrowed rotations at appropriate nodes.



3 Graph (20 points) (Learning Objectives 4 and 5)

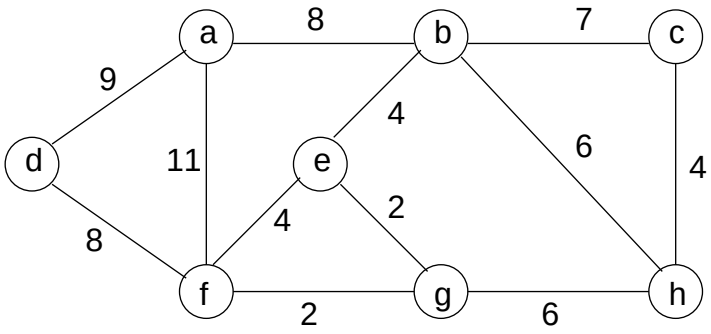
(a) (6 points) Consider the depth-first search algorithm that we covered in class.

```
1  DFS(G):
2     for each node u in V[G] {
3        color[u] = WHITE;
4        parent[u] = NULL;
5     }
6     time = 0;
7     for each node u in V[G]
8        if (color[u] == WHITE)
9           dfs(G, u);
```

```
10 dfs(G, u):
11    color[u] = GRAY;
12    time = time + 1;
13    d[u] = time;
14    for each v in adj[u]
15       if (color[v] == WHITE) {
16          parent[v] = u;
17          dfs(G, v);
18       }
19    time = time + 1;
20    f[u] = time;
21    color[u] = BLACK;
```

Apply the DFS algorithm on the weighted undirected graph `G(V,E)` in the next page and complete the table for `d[·]`, `f[·]`, and `parent[·]` of every node in the graph. Assume that the nodes are arranged in alphabetical order and each adjacency list is also arranged in alphabetical order.

When you apply DFS on an undirected graph, even though an undirected edge (u,v) (or (v,u)) is incident on both nodes u and v, the edge will be visited only once. We treat it as a directed edge <u,v> from u to v or a directed edge <v,u> from v to u according to whichever of <u,v> or <v,u> is encountered first. For the DFS on the given graph, for example, the undirected edge (a,b) (or (b,a)) is treated as a directed edge <a,b> from a to b.



	a	b	c	d	e	f	g	h
d[·]	1	2						
f[·]								
parent[·]	NULL	a						

(b) (4 points) Applying the DFS algorithm on an undirected graph categorizes the edges into **tree** edges or **back** edges. Identify in the graph in **Question 3(a)** the **back** edges.

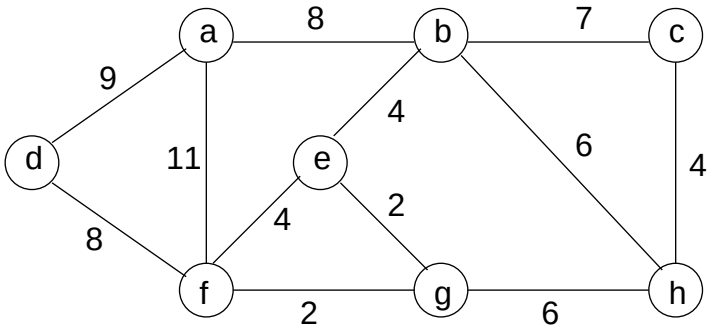
(c) (4 points) Suppose in the process of applying DFS, you encounter a back edge <u,v> (derived from an undirected edge (u,v) or (v,u)). Write down the pseudo-code to identify and return the (directed) edge with the largest weight that forms a cycle with <u,v> in the graph. Note that <u,v> may be the edge with the largest weight. Assume that w<u,v> is the weight of an edge, and that given u and v, you can get that in $O(1)$ time complexity. Also assume that you have access to d[·], f[·], and parent[·] computed so far. The time complexity of the algorithm should be $O(|V|)$.

```
Find_edge_with_largest_weight_in_cycle(G, <u,v>):
```

(d) (6 points) Suppose we modify the DFS algorithm in **Question 3(a)** such that it uses the algorithm in **Question 3(c)** to identify and return the edge with the largest weight in the **first** cycle encountered in depth first search of an undirected graph `G(V,E)`. If the graph has no cycles, it returns `NULL`. Let's call this algorithm `DFS_find_largest_edge_in_first_cycle(G)`. We use this algorithm in the following pseudo-code:

```
Mysterious(G):
   // while there exists a cycle
   while ((e = DFS_find_largest_edge_in_first_cycle(G)) != NULL) {
      Let e' be the undirected version of e
      E[G] = E[G] - {e'} // remove e' from E[G]
   }
   return G;
```

Identify, **in order**, the **first two edges** that are removed from `G(V,E)` when you run the `Mysterious` algorithm on the graph given in **Question 3(a)**. For your convenience, the graph is re-printed here. You may cross out the removed edges in the diagram and use a number beside each removed edge to indicate the ordering.



What problem can the `Mysterious(G)` algorithm solve, or equivalently, what does the `Mysterious(G)` algorithm compute?