ECE368 Exam 1
Spring 2009

Wednesday, February 25, 2009
6:30-7:30pm
MSEE B012

READ THIS BEFORE YOU BEGIN

This is a *closed book* exam. Calculators are not needed but they are allowed. Since time allotted for this exam is *exactly* 60 minutes, you should work as quickly and efficiently as possible. *Note the allotment of points and do not spend too much time on any problem*.

Always show as much of your work as practical–partial credit is largely a function of the *clarity and quality* of the work shown. Be *concise*. It is fine to use the blank page opposite each question for your work.

This exam consists of 10 pages (including this cover sheet and a grading summary sheet at the end); please check to make sure that *all* of these pages are present *before* you begin. Credit will not be awarded for pages that are missing – it is *your responsibility* to make sure that you have a complete copy of the exam.

IMPORTANT: Write your login at the TOP of EACH page. Also, be sure to *read and sign* the *Academic Honesty Statement* that follows:

"In signing this statement, I hereby certify that the work on this exam is my own and that I have not copied the work of any other student while completing it. I understand that, if I fail to honor this agreement, I will receive a score of ZERO for this exercise and will be subject to possible disciplinary action."

`Printed Name:`

`login:`

`Signature:`

DO NOT BEGIN UNTIL INSTRUCTED TO DO SO ...

1 (25 points) Consider the following adaptive bubble sort algorithm, which we covered in class, applied to array `r[0..`$n-1$`]` with n integers:

```
max_i ← $n-2$
do {
  last_swap ← $-1$
  for i ← 0 to max_i {
    if r[i] > r[i+1] {
      r[i] ↔ r[i+1]
      last_swap ← i
    }
  }
  max_i ← last_swap − 1
} while last_swap ≥ 0
```

In the algorithm, the position of the last occurrence of data swap in each pass is used to improve the run-time of the bubble sort algorithm. Note that a pass "bubbles" the largest element in the unsorted subarray to the correct position.

a. (6 points) Modify the preceding pseudo-code to also detect the position of the first occurrence of data swap in each pass. Use `first_swap` to store that position in your pseudo-code.

b. (6 points) What can you say about the "sortedness" of the subarray `r[0..first_swap − 1]`, assuming that `first_swap` > 0?

c. (7 points) Modify the preceding pseudo-code to take advantage of the computed `first_swap` to reduce the number of comparisons in a bubble sort algorithm. (If you have difficulty in computing `first_swap` in **(a)**, assume that `first_swap` is available.)

d. (6 points) Does the new bubble sort that you have written reduce the number of swaps when compared to the original bubble sort? Justify your answer.

2. (25 points)
a. (5 points) True or False: $(n+1)! = O(n!)$? Justify your answer.

b. (5 points) True or False: $2^{n+1} = O(2^n)$? Justify your answer.

c. (5 points) True or False: $n^{1/\log n} = O(2^{\log n})$? Justify your answer.

d. (10 points) Consider the following instructions:

```
1.    for $i \leftarrow 0$ to $n$
2.        for $j \leftarrow 0$ to $n - i$
3.            for $k \leftarrow 0$ to $n - j$
4.                $A[k] \leftarrow A[k] - A[i]$;
```

Write down the respective numbers of times instructions 1 and 4 are executed in terms of i, j, k, and n. Do not attempt to evaluate/expand your expressions.

Instruction 1:

Instruction 4:

3. (25 points) Consider an array-based implementation of singly linked list(s). The contents of the array are as follows:

`index`	`key`	`next_index`
9	A	0
8	B	-1
7	C	3
6	D	8
5	E	7
4	F	-1
3	G	-1
2	H	6
1	I	2
0	J	4

The array contains three lists:

- `Unused_List`: Maintains a list of unused objects. The list starts at index 9.
- `Stack_List`: Implements a stack. The list starts at index 1.
- `Queue_List`: Implements a queue. The list starts at index 5.

a. (6 points) Write down the keys (from the head to the tail) in the three lists.

- `Unused_List`:
- `Stack_List`:
- `Queue_List`:

b. (3 points) The primitive operations of queue abstract data type are `Empty`(Q), `Enqueue`(Q, *element*), `Dequeue`(Q), `Front`(Q), and `Rear`(Q).

The primitive operations of stack abstract data type are `Empty`(S), `Stack_Top`(S), `Push`(S, *element*), and `Pop`(S).

You want to implement all the stack and queue primitives in $O(1)$ time complexity. Which is the `Stack_Top` of the stack abstract data type, the head or tail of `Stack_List`?

Which is the `Front` of the queue abstract data type, the head or tail of `Queue_List`? What about the `Rear`?

c. (4 points) In order to implement all the stack and queue primitives in $O(1)$ time complexity, what other indices, besides the indices pointing to the heads of the lists, do you have to keep track? (Put 'None' if you do not require additional indices.) The goal here is to minimize the additional space requirement. Note that for `Unused_List`, you maintain only the index that points to the head of the list.

- `Stack_List`:
- `Queue_List`:

d. (12 points) Show the contents of the array after performing each of the following two consecutive operations:

1. `Enqueue(Queue_List, Stack_Top(Stack_List))`
2. `Dequeue(Queue_List)`

Update also the head indices of the lists, as well as the indices you have added in **(c)**.

Note that an *element* is stored as `key`, and is returned by the following operations: `Dequeue`(Q), `Front`(Q), `Rear`(Q), `Stack_Top`(S), and `Pop`(S).

	Original		Enqueue		Dequeue	
`index`	`key`	`next_index`	`key`	`next_index`	`key`	`next_index`
9	A	0				
8	B	-1				
7	C	3				
6	D	8				
5	E	7				
4	F	-1				
3	G	-1				
2	H	6				
1	I	2				
0	J	4				

Head index	Original	Enqueue	Dequeue
`Unused_List`	9		
`Stack_List`	1		
`Queue_List`	5		

Other indices	Original	Enqueue	Dequeue

4 (25 points)
a. (10 points) Consider a data structure covered in Lecture 11, in which the items from a linked list are stored as leaf nodes in a binary tree. The non-leaf nodes store the number of elements in the left subtree. The following recursive routine given in the lecture finds the *k*th element in the linked list, assuming that the root node is not NULL and the linked list contains at least *k* elements:

```
Find_Kth(Node, k):
1.      if (Empty(Left_Child(Node)) && Empty(Right_Child(Node)))
2.        return Info(Node)
3.      if (k ≤ Count(Node))
4.        return Find_Kth(Left_Child(Node), k)
5.      else
6.        return Find_Kth(Right_Child(Node), k − Count(Node))
```

Modify the routine to perform the search in an iterative fashion.

b. (15 points) Consider the following breadth-first traversal routine using a queue Q to print all nodes in a binary tree with $O(n)$ time complexity, where n is the number of nodes in the binary tree:

```
Breadth_First_Traversal(Tree_root):
    Enqueue($Q$, Tree_root)
    while ($Q$ not empty) {
      node $\leftarrow$ Dequeue($Q$)
      if (node != NULL) {
        print node
        Enqueue($Q$, left(node))
        Enqueue($Q$, right(node))
      }
    }
```

The given breadth-first traversal routine prints the nodes in a top-down fashion. Nodes are printed level-by-level, starting from the level 0, where the root node is.

Now, suppose you have to modify the given routine to print the nodes in a bottom-up fashion. In particular, the new routine should reverse the order in which the nodes are printed in the given routine. Moreover, you are not allowed to use recursion in the modified pseudo-code.

With the aid of an *additional* abstract data type, write down an $O(n)$ routine to print the nodes in a bottom-up fashion. Specify the additional abstract data type. Also suggest another abstract data type that you may use to perform the same function.

Question	Max	Score
1	25	
2	25	
3	25	
4	25	
Total	100	