ECE368 Exam 2
Spring 2011

Thursday, April 13, 2010
6:30-7:30pm
PHYS 203

READ THIS BEFORE YOU BEGIN

This is an *open-book, open-notes* exam. Electronic devices are not allowed. The time allotted for this exam is *exactly* 60 minutes. *It is in your best interest to 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 (or at the back of each question) for your work. Do draw an arrow to indicate that if you do so.

This exam consists of 9 pages; 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."

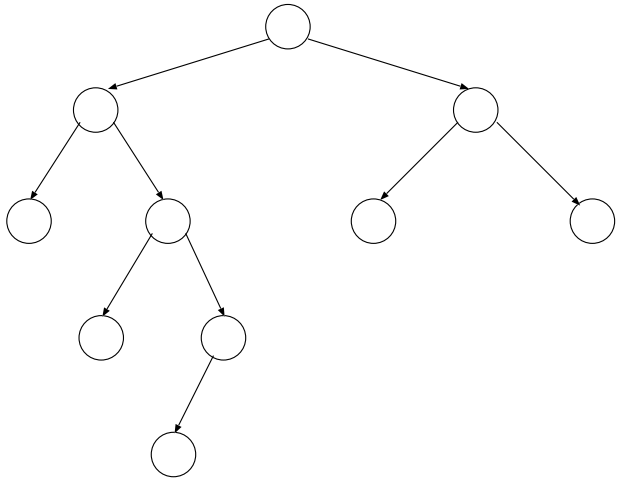
`Printed Name:`

`login:`

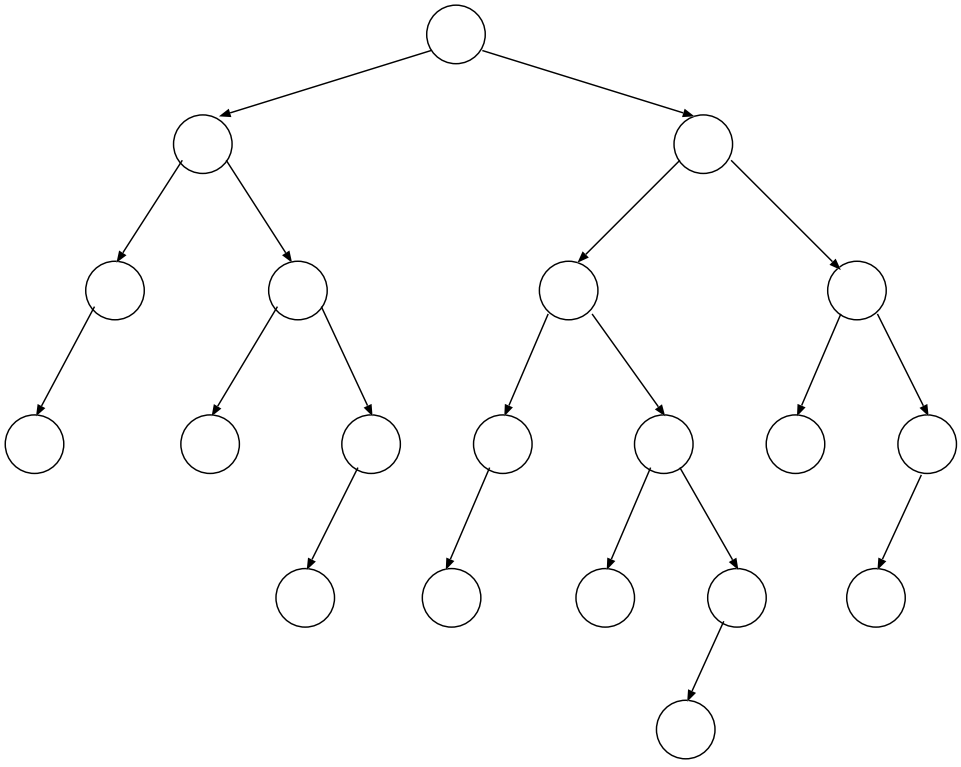
`Signature:`

DO NOT BEGIN UNTIL INSTRUCTED TO DO SO ...

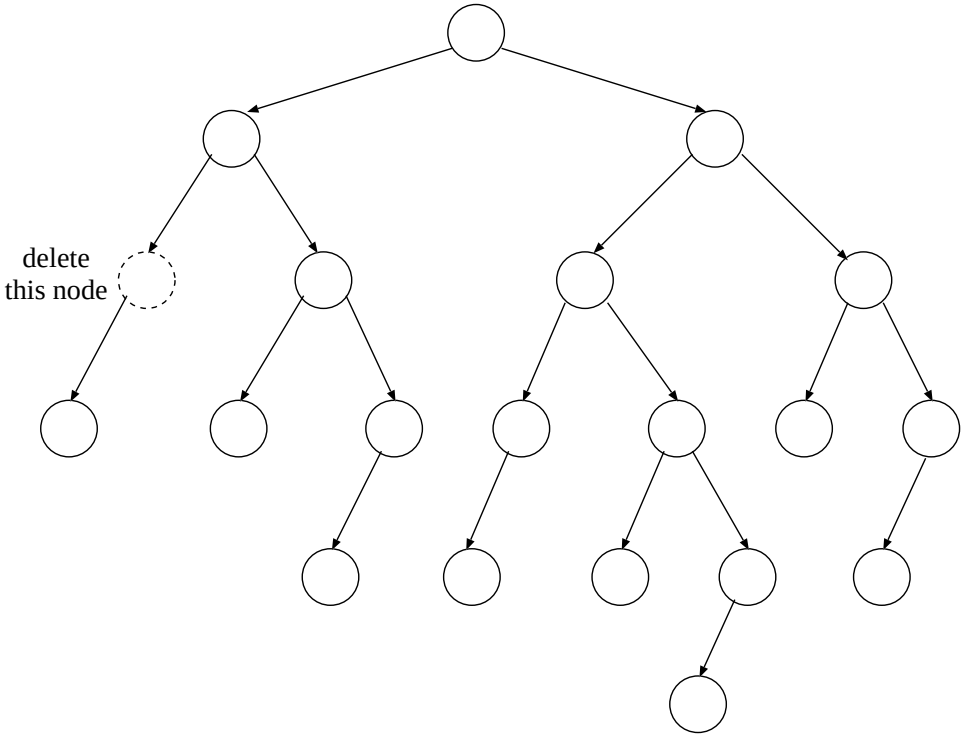
1. (30 points) (a) (10 points) Consider the following tree. Following the definition from the notes that the "balance" of a node is "height of left subtree" − "height of right subtree", write down the balance of each node in the following tree. Add to the tree the fewest number of nodes to make it height-balanced.



(b) (8 points) Consider the following height-balanced binary search tree. The twenty distinct keys that are stored in this binary search are $\{1, 2, \ldots, 20\}$. Write down these keys in each node (circle) of the binary search tree.



(c) (12 points) Consider the same height-balanced binary search tree in **(b)**. Delete the node highlighted below. Performs necessary rotation(s) to maintain the height-balanceness. Draw the final height-balanced binary search tree after deletion and necessary rotation(s).



2. (30 points) (a) (15 points) The following routines perform an even partitioning of an array, followed by merging, i.e., the array is divided into two subarrays, and `Rec_Mergesort` is recursively applied on the two subarrays.

```
Merge(r[], tmp[], lb, mid, ub):
   i ← lb
   j ← mid+1
   for m ← lb to ub {
      if (i > mid) r[m] ← tmp[j++]
      else if (j > ub) r[m] ← tmp[i++]
      else if (tmp[j] < tmp[i]) r[m] ← tmp[j++]
      else r[m] ← tmp[i++]
   }

Rec_Mergesort(r[], tmp[], lb, ub):
1.    if lb ≥ ub
2.      return
3.    mid ← (lb+ub)/2  // Integer division
4.    Rec_Mergesort(tmp, r, lb, mid)   // Store sorted subarrays in tmp[]
5.    Rec_Mergesort(tmp, r, mid+1, ub) //   using r[] as auxiliary
6.    Merge(r, tmp, lb, mid, ub)  // Merge sorted subarrays tmp[] into r[]
```

We use the preceding subroutines to perform Mergesort on an array $\mathtt{r}[0..\mathtt{n}-1]$ as follows:

```
Mergesort(r[], size):
   allocate space for tmp[]
   for i ← 0 to n−1 {
      tmp[i] ← r[i] // Copy r[] to tmp[]
   }
   Rec_Mergesort(r, tmp, 0, n−1)
```

Consider the application of `Mergesort` on the array $\mathtt{r} = [16,4,1,3,2,17,10,9,14,8,7]$ with `Mergesort(r, 11)`, which will call `Rec_Mergesort(r, tmp, 0, 10)`. in `Rec_Mergesort(r, tmp, 0, 10)`. Write down the contents of `r[]` and `tmp[]` immediately after the complete execution of the first recursive call of `Rec_Mergesort` (line 4) in `Rec_Mergesort(r, tmp, 0, 10)`.

Write down the contents of `r[]` and `tmp[]` immediately after the complete execution of the second recursive call of `Rec_Mergesort` (line 5) in `Rec_Mergesort(r, tmp, 0, 10)`.

(b) (10 points) Believing that recursions should be removed, Prof. Koh wrote the following routine to perform merge sort of $\mathtt{r}[0..\mathtt{n}-1]$ using iterations:

```
 Iter_Mergesort(r[], tmp[], n):
01.    sorted ← r // Array r[] contains sorted 1-element subarrays
02.    merged ← tmp // Use tmp[] to store the merged results in the first pass
03.    size ← 1
04.    while size < n {
05.      i ← 0
06.      while i < n − size { // Merge when there is a right subarray
07.        Merge(merged, sorted, i, i+size-1, min(i+2*size-1, n-1))
08.        i ← i + 2*size
09.      }
10.      sorted ↔ merged // Exchange roles in the next pass
11.      size ← 2*size
12.    }
13.    if sorted == tmp { // tmp contains sorted array
14.      for i ← 0 to n−1 {
15.        r[i] ← tmp[i] // Copy tmp[] to r[]
16.      }
17.    }
```

He used the preceding subroutine (and the `Merge` subroutine in **(a)**) to perform Mergesort on an array $\mathtt{r}[0..\mathtt{n}-1]$ as follows:

```
Mergesort(r[], n):
   allocate space for tmp[]
   for i ← 0 to n−1 {
     tmp[i] ← r[i] // Copy r[] to tmp[]
   }
   Iter_Mergesort(r, tmp, n)
```

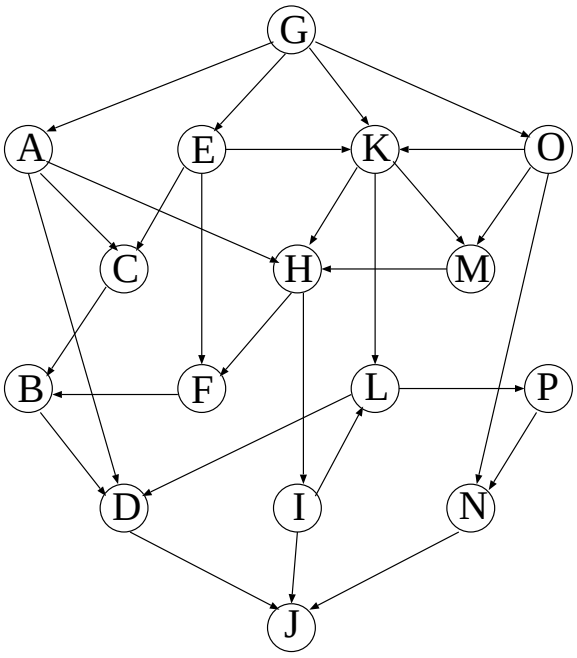
Write down the contents of `r[]` and `tmp[]` immediately after the program exits from the outer while-loop (`while size < n`) (line 12) of `Iter_Mergesort` when `Mergesort` is applied on $\mathtt{r} = [16, 4, 1, 3, 2, 17, 10, 9, 14, 8, 7]$.

(c) (5 points) Is Prof. Koh's implementation of merge sort using iterations in **(b)** correct? If yes, justify your answer. If not, amend his pseudo-code such that merge sort using iterations can be performed correctly. For your convenience, the routines `Iter_Mergesort` and `Mergesort` are replicated here.

```
Iter_Mergesort(r[], tmp[], n):
  sorted ← r; // Array r[] contains sorted 1-element subarrays
  merged ← tmp; // Use tmp[] to store the merged results in the first pass
  size ← 1
  while size < n {
    i ← 0
    while i < n − size { // Merge when there is a right subarray
      Merge(merged, sorted, i, i+size-1, min(i+2*size-1, n-1))
      i ← i + 2*size
    }
    sorted ↔ merged // Exchange roles in the next pass
    size ← 2*size
  }
  if sorted == tmp { // tmp contains sorted array
    for i ← 0 to n−1 {
      r[i] ← tmp[i] // Copy tmp[] to r[]
    }
  }

Mergesort(r[], n):
  allocate space for tmp[]
  for i ← 0 to n−1 {
    tmp[i] ← r[i] // Copy r[] to tmp[]
  }
  Iter_Mergesort(r, tmp, n)
```

3. (30 points) (a) (10 points) Consider the following directed acyclic graph. Which representation, adjacency matrix or adjacency list, would you use to implement depth first search? Justify your answer.

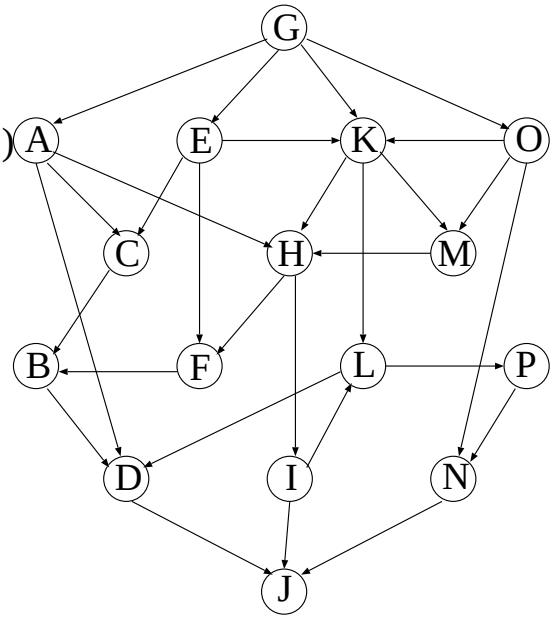


Show your chosen representation of the given graph (**Do not show both representations**):

- If your choice is a adjacency matrix representation, use '1' to indicate the presence of an edge, and '0' to indicate the absence of an edge. The rows and columns of the adjacency matrix should be in alphabetical order. *Show only the first 8 rows of the matrix.*
- If your choice is an adjacency list representation, the vertices should be listed in alphabetical order in the array representing the nodes and also in the linked lists representing the edges. *Show only the first 8 lists in the array.*

(b) (10 points) Perform a depth first search on the given graph. Assume that the vertices are sorted in alphabetical order in the array representing the nodes and also in the linked lists representing the edges. Indicate beside each node the start and end time stamps ($d[\cdot]$ and $f[\cdot]$ in the pseudo-code given in the lecture notes) when the node was traversed (or was on the code stack). The parentheses beside node *A*, for example, can be used to record the start and end time stamps of *A*.

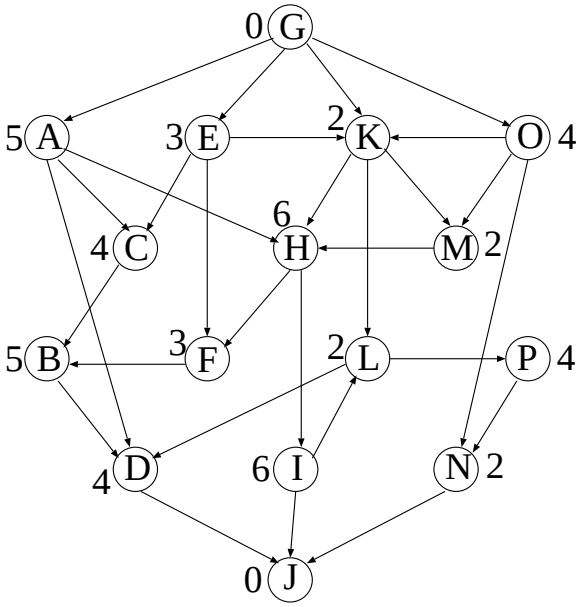
(,)



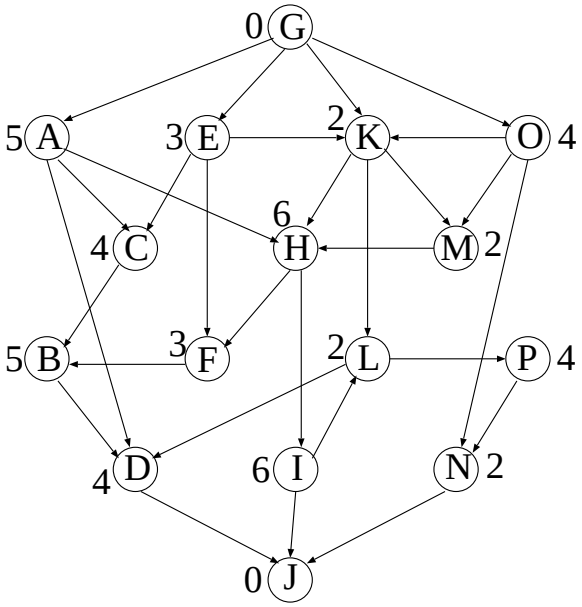
Show the ordering of nodes produced when you run the depth-first-search-based topological sort algorithm on the graph.

(c) (10 points) The given graph represents a project with various tasks depicted by the circles (nodes). The time it takes to complete each task is shown beside each node below. A task cannot start unless all its predecessors have been completed. The project starts with task G and ends with task J. Determine the earliest finish time of the entire project and the slack of each task. The graph is replicated for your convenience.

Earliest finish time:



Node	Slack
A	
B	
C	
D	
E	
F	
G	
H	
I	
J	
K	
L	
M	
N	
O	
P	



Question	Max	Score
1	30	
2	30	
3	30	
Total	90	