ECE368 Exam 2
Spring 2014

Thursday, April 17, 2014
6:30-7:30pm
PHYS 112

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.

Learning objective 1 can be satisfied if you score 10 points or more in question 1 or question 3. Learning objective 2 can be met if you obtain 10 points or more in question 3. Learning objective 3 can be achieved by scoring 10 points or more in question 2 or question 3. Learning objective 4 can be satisfied if you obtain 10 points or more in question 3.

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 exam and will be subject to possible disciplinary action."

`Printed Name:`

`login:`

`Signature:`

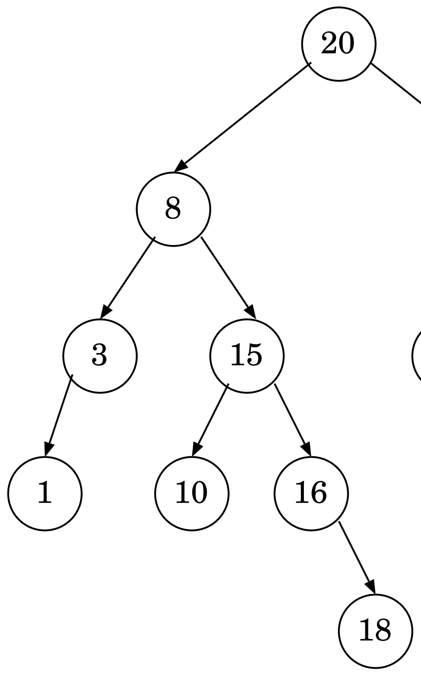
DO NOT BEGIN UNTIL INSTRUCTED TO DO SO ...

1. (20 points) Height-Balanced Binary Search Trees.
(a) (10 points) Starting from an empty (height-balanced) binary search tree, perform the insertion of the following integers in succession:

100, 200, 150, 125, 112, 50

If the insertion results in an unbalanced tree, you have to perform rotation(s) to balance the tree. *Draw the height-balanced binary search tree after each insertion.*

(b) (10 points) You are given the following height-balanced binary search tree:



For each key in the tree, indicate the number of rotations that are required to be performed if that key is to be deleted from the tree. If the deletion involves a node that has two child nodes, replace the key in that node with its immediate in-order *predecessor* (the key right before the deleted key in an in-order traversal), and delete the node containing the immediate in-order predecessor instead. *You do not have the draw the height-balanced binary search tree after the deletion operation.*

1:	3:
8:	10:
15:	16:
18:	20:
50:	58:
60:	63:

2. (20 points) Mergesort Algorithms.
(a) (8 points) The following routines divide an array into two subarrays, apply `Rec_Mergesort` recursively on the two subarrays, and then merge the two sorted subarrays.

```
Merge(r[], tmp[], lb, mid, ub):
  i ← lb;
  j ← mid+1;
  for m ← lb to ub {
    if (i > mid) { r[m] ← tmp[j]; j++; }
    else if (j > ub) { r[m] ← tmp[i]; i++; }
    else if (tmp[j] < tmp[i]) { r[m] ← tmp[j]; j++; }
    else { r[m] ← tmp[i]; i++; }
  }
Rec_Mergesort(r[], tmp[], lb, ub):  // Store sorted array in r[]
                                    // using tmp[] as auxiliary
  if lb ≥ ub { return; }
  mid ← (lb+ub)/2;  // Integer division
  Rec_Mergesort(tmp, r, lb, mid);    // Store sorted subarrays in tmp[]
  Rec_Mergesort(tmp, r, mid+1, ub); //   using r[] as auxiliary
  Merge(r, tmp, lb, mid, ub);  // Merge sorted subarrays tmp[] into r[]
```

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

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

Draw the computation tree where each node corresponds to the function `Rec_Mergesort` when we call `Mergesort(r, 11)`, where $\texttt{r} = [10,4,1,3,2,17,16,9,14,8,7]$. In each node of the computation tree, you should write down the corresponding input parameters. The input parameters for the root node of the computation tree are $(\texttt{r},\texttt{tmp},0,10)$.

(b) (12 points) We also introduced in the class bitonic merging to merge two subarrays, of which the first is in ascending order and the second is in descending order.

(i) (4 points) Here, we show a bitonic merging routine that produces an ascending array in `r[]` from two such subarrays (i.e., first is ascending and second is descending) stored in `tmp[]`.

```
Ascending_Bitonic_Merge(r[], tmp[], lb, ub):
  i ← lb;
  j ← ub;
  for m ← lb to ub {
    if (tmp[j] < tmp[i]) { r[m] ← tmp[j]; j--; }
    else { r[m] ← tmp[i]; i++; }
  }
```

Write in the following a bitonic merging routine that produces a *descending* array in `r[]` from two such subarrays (i.e., first is ascending and second is descending) stored in `tmp[]`:

```
Descending_Bitonic_Merge(r[], tmp[], lb, ub):
  i ← lb;
  j ← ub;
```

(ii) (8 points) We would like to apply bitonic merging to perform Mergesort. Here is one version of the new Mergesort routine, called `Rec_Bitonic_Mergesort`, where an additional input parameter `direction` is used to indicate whether the sorted array `r[]` should be `ascending` or `descending`:

```
 Rec_Bitonic_Mergesort(r[], tmp[], lb, ub, direction):
1.   if lb ≥ ub { return; }
2.   mid ← (lb+ub)/2;
3.   Rec_Bitonic_Mergesort(tmp, r, lb, mid, ascending); // first subarray ascending
4.   Rec_Bitonic_Mergesort(tmp, r, mid+1, ub, descending); // second subarray
                                                           // descending
5.   if direction == ascending { // Merge sorted subarrays in ascending order
6.     Ascending_Bitonic_Merge(r, tmp, lb, ub);
7.   } else { // Merge sorted subarrays in descending order
8.     Descending_Bitonic_Merge(r, tmp, lb, ub);
9.   }
```

We now perform Mergesort on an array r[0..n − 1] as follows:

```
Bitonic_Mergesort(r[], n):
  allocate space for tmp[];
  for i ← 0 to n − 1 {
    tmp[i] ← r[i]; // Copy r[] to tmp[]
  }
  Rec_Bitonic_Mergesort(r, tmp, 0, n−1, ascending);
```

Consider the application of `Bitonic_Mergesort` on the array $r = [10, 4, 1, 3, 2, 17, 16, 9, 14, 8, 7]$ with `Bitonic_Mergesort(r, 11)`, which calls `Rec_Bitonic_Mergesort(r, tmp, 0, 10, ascending)`. Write down the contents of `r[]` and `tmp[]` immediately after the complete execution of the recursive call of `Rec_Bitonic_Mergesort` in line 3 of `Rec_Bitonic_Mergesort(r, tmp, 0, 10, ascending)`.

Write down the contents of `r[]` and `tmp[]` immediately after the complete execution of the recursive call of `Rec_Bitonic_Mergesort` in line 4 of `Rec_Bitonic_Mergesort(r, tmp, 0, 10, ascending)`.

3. (20 points) Graphs and Minimum-Cost Spanning Trees. We have learned in a homework exercise a greedy algorithm of finding a minimum-cost tree that spans an undirected graph $G(V,E)$, i.e., a tree that contains all nodes in the graph, and the tree edges connecting all the nodes have the least total weight.

The basic idea is to process the edges in a non-increasing order. For each edge, we ask whether the removal of the edge will result in a disconnected graph. If the graph remains connected after the removal, the edge can be safely discarded from the minimum-cost spanning tree. Otherwise, the edge should be kept in the minimum-cost spanning tree. The pseudo-code is given below.

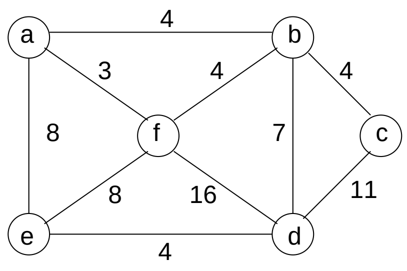
```
 Greedy-MST(G, V, E):
1.   MST ← G(V,E); /* MST is G to begin with */
2.   Build a priority queue PQ of edges in E[G] using max-heap;
3.   while PQ is not empty {
4.     e ← Dequeue(PQ); /* edge e with highest weight in PQ is dequeued */
5.     MST' ← MST - e; /* temporarily remove e from MST */
6.     if MST' is connected
7.       MST ← MST'; /* e can be safely removed */
8.   }
9.   return MST
```

Here, a max-heap is used to implement the priority queue PQ. In such an implementation, you use an array to store the weighted edges and also a field to indicate the size of the heap. In the case of building a *max-heap* of n integers stored in array $r[0..n-1]$, the following routines (taken from the lecture notes) are used:

```
// build a max-heap of integers
for i ← n/2 - 1 down to 0
  Downward_heapify(r[], i, n-1);
```

```
// heapify r[i] downward in r[i..ub]
Downward_heapify(r[], i, ub):
  temp_r ← r[i];
  while (2*i+1 <= ub) {
    j ← 2*i + 1;
    if j < ub and r[j] < r[j+1] {
      j ← j+1;
    }
    if temp_r ≥ r[j] {
      break;
    } else {
      r[i] ← r[j];
      i ← j;
    }
  }
  r[i] ← temp_r;
```

(a) (5 points) The `Greedy-MST` function is applied to this undirected graph:



Assume that the edges are initially stored in an array with the following order:

(a,b), (a,e), (a,f), (b,c), (b,d), (b,f), (c,d), (d,e), (d,f), (e,f).

Show the ordering of the edges (**not** their weights) in the array of the *max-heap* data structure constructed in line 2 of `Greedy-MST` using an approach similar to that for the construction of a max-heap of integers with `Downward_heapify`.

(b) (5 points) Line 6 of `Greedy-MST` requires you to check whether `MST'` is connected, meaning that starting from any arbitrary node in `MST'`, you can still reach all other nodes in `MST'`. Consider the following the depth-first-search algorithm:

```
DFS(G, V, E):
  for each node u in V[G]
    color[u] ← WHITE
  time ← 0
  for each node u in V[G]
    if (color[u] == WHITE)
      dfs(G, V, E, u)
```

```
dfs(G, V, E, u):
  color[u] ← GRAY
  time ← time + 1
  d[u] ← time
  for each v in adj[u]
    if (color[v] == WHITE)
      dfs(G, V, E, v)
  time ← time + 1;
  f[u] ← time;
  color[u] ← BLACK
```

Modify the `DFS(G, V, E)` algorithm to check "`if MST' is connected`" in $O(V+E)$ time complexity. Of course, you should be passing `MST'` to the function to check for connectivity. For simplicity, you may assume that the undirected edges are each implemented as two directed edges.

(c) (5 points) What is the *overall* worst-case time complexity of each of the following statements in `Greedy-MST`, i.e., the time complexity for executing the statement in the *entire algorithm*?

- Line 2:

- Line 4:

- Line 6:

(d) (5 points) Instead of the condition `‘‘PQ is not empty’’` in line 3, there are other conditions that you can use to safely terminate the `while`-loop. Use a condition that would allow you to terminate as early as possible, i.e., you do not have to empty `PQ`.

The evaluation of the new condition (and the associated operations required for such evaluation) must be equally simple (in terms of time complexity and space complexity) as the evaluation of the condition `‘‘PQ is not empty’’`.

With the use of the new condition, how many iterations of the instructions in lines 4–7 would be executed for a given connected undirected graph $G(V,E)$ in the best case scenario? In other words, what is the fewest number of iterations that the instructions in lines 4–7 would be executed?