ECE368 Exam 2
Spring 2013

Wednesday, April 10, 2013
6:30-7:30pm
EE 170

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 8 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:`

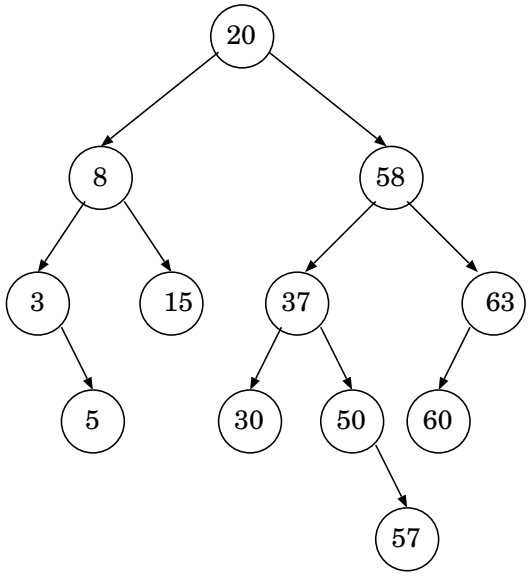
`login:`

`Signature:`

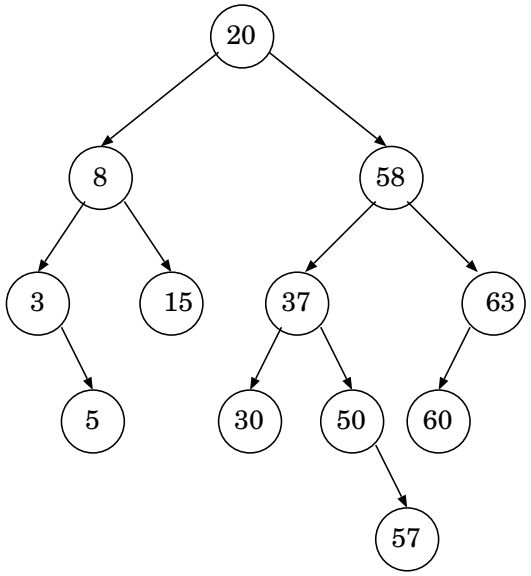
DO NOT BEGIN UNTIL INSTRUCTED TO DO SO ...

1. (20 points) Height-Balanced Binary Search Trees. Each of the following questions shows a height-balanced binary search tree. You are asked to perform either an insertion or a deletion operation. If the deletion involves a node that has two children, 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. Draw the height-balanced binary search tree after the insertion operation or the deletion operation (and after performing the necessary rotation operation(s) to maintain a balanced height). Clearly state the rotation operations that you have used.

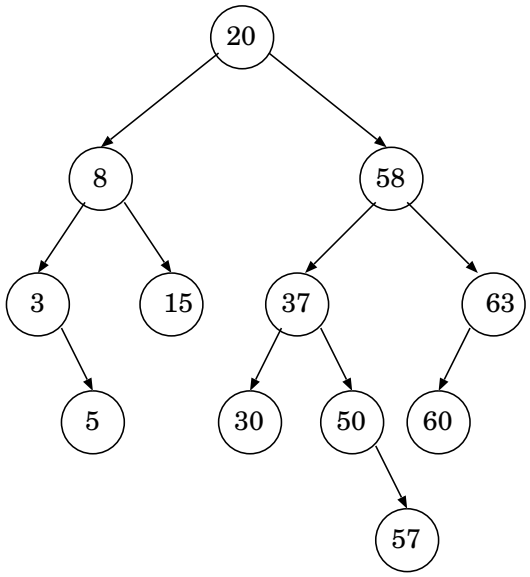
(a) (5 points) Insert 10.



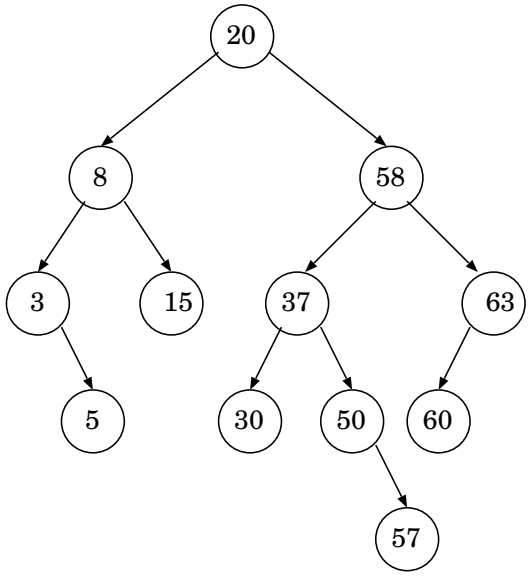
(b) (5 points) Insert 4.



(c) (5 points) Delete 63.



(d) (5 points) Delete 20.

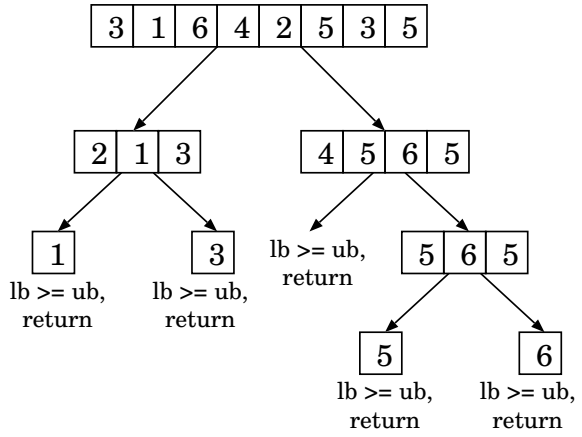


2. (20 points) Quicksort. In the class, we perform partitioning and quicksort as follows:

```
Partition_lec(r[], lb, ub):
  pivot ← r[lb];
  down ← lb;
  up ← ub;
  while down < up {
    while (r[down] ≤ pivot and down < ub) { down++; }
    while r[up] > pivot { up--; }
    if down < up {
      r[down] ↔ r[up];
    }
  }
  r[lb] ← r[up];
  r[up] ← pivot;
  return up;

Quicksort_lec(r[], lb, ub):
  if lb ≥ ub {
    return;
  }
  pivot_idx ← Partition_lec(r[], lb, ub);
  Quicksort_lec(r[], lb, pivot_idx-1);
  Quicksort_lec(r[], pivot_idx+1, ub);
```

The following computation tree shows the subarrays that are recursively operated on by the `Quicksort_lec` routine when the routine was invoked with an array of $[3,1,6,4,2,5,3,5]$.



Now, Prof. Koh modified the `Partition_lec` routine in order to use the mean (or average) of `r[lb..ub]` as the pivot, which is of type `float`. He also modified the `Quicksort_lec` routine to use the new partitioning algorithm. The two modified routines are in the next page.

```
Partition_lec_mean(r[], lb, ub):
  sum ← 0;
  for i ← lb to ub { sum ← sum + r[i]; }
  pivot ← sum / (ub - lb + 1); /* floating point division */;
  down ← lb;
  up ← ub;
  while down < up {
    while (r[down] ≤ pivot and down < ub) { down++; }
    while r[up] > pivot { up--; }
    if down < up {
      r[down] ↔ r[up];
    }
  }
  return up;

Quicksort_lec_mean(r[], lb, ub):
  if lb ≥ ub {
    return;
  }
  pivot_idx ← Partition_lec_mean(r[], lb, ub);
  Quicksort_lec_mean(r[], lb, pivot_idx);
  Quicksort_lec_mean(r[], pivot_idx + 1, ub);
```

(a) (10 points) Draw the computation tree showing the subarrays that are recursively operated on when the `Quicksort` routine is called with the array $[3,1,5,6,3,3,4,2,3,5]$.

(b) (10 points) Identify the problem encountered in this case. *Without adversely affecting the runtime*, make changes to the `Quicksort_lec_mean` routine above to eliminate the problem, while still using the routine `Partition_lec_mean` to perform partitioning based on the mean of the given array.

3. (20 points) Graphs and Minimum-Cost Spanning Trees.

We have learned in the class (not in lecture notes) 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.

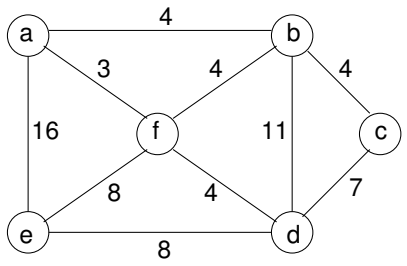
```
 Greedy-MST(G, V, E):
1.    MST ← G(V,E); /* MST is G to begin with */
2.    Build a priority queue PQ of edges in E[G] using max-heap;
3.    while PQ is not empty {
4.      e ← Dequeue(PQ); /* edge e with highest weight in PQ is dequeued */
5.      MST' ← MST - e; /* temporarily remove e from MST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

The `Greedy-MST` function is applied to this undirected graph:



(i) (5 points) 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`.

(ii) (5 points) How would you implement `Dequeue(PQ)` efficiently? What is the time complexity of the operation `Dequeue(PQ)`?

(iii) (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.

(iv) (5 points) What is the *overall* 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: