ECE368 Exam 2
Spring 2012

Tuesday, April 10, 2012
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.

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:`

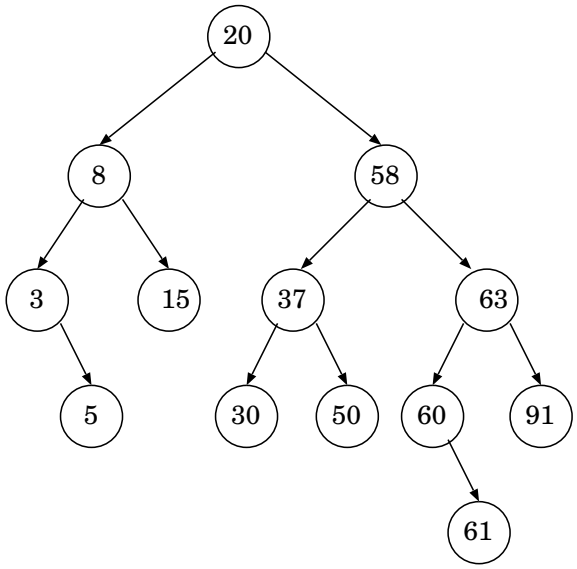
The 26 letters in alphabetical order: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

DO NOT BEGIN UNTIL INSTRUCTED TO DO SO ...

1. (20 points) Height-Balanced Trees.

(a) (14 points) The following diagram shows a height-balanced binary search tree (AVL-tree). Perform the following operations: Insert 62 and delete 20. 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 respective height-balanced binary search trees after the insertion operation and the deletion operation (and after performing the necessary rotation operations). Clearly state the rotation operations that you have used.



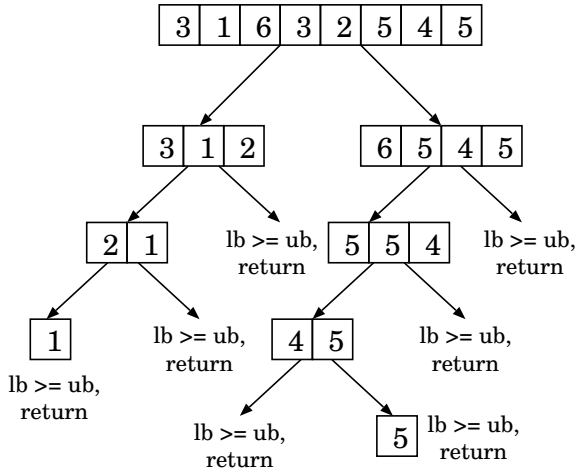
(b) (6 points) Draw a height-balanced tree that has the smallest height and the fewest nodes such that the deletion of some node in that tree would require 4 rotations. Identify the node that has to be deleted to trigger the 4 rotations, as well as the 4 rotations.

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 $[3,1,6,3,2,5,4,5]$.



This page contains only information and no questions.

(a) (10 points) Now, assume that you are given an array of *distinct* integers. The partitioning algorithm has been modified to use the mean (or average) of `r[lb..ub]` as the pivot, which is of type `float`. (This is not the most efficient way to use the mean of an array for partitioning.) Complete the pseudo-code of a new quicksort algorithm based on the modified partitioning algorithm.

```
Partition(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(r[], lb, ub):
  if lb ≥ ub {
    return;
  }
  pivot_idx ← Partition(r[], lb, ub);
  Quicksort(r[],            ,            ); /* fill in the parameters */
  Quicksort(r[],            ,            ); /* fill in the parameters */
```

Based on the inequality relationship between the elements in `r[lb,up]` and `r[up+1,ub]` when the `Partition` routine exits from the outer while-loop, fill in the appropriate parameters to be passed into the `Quicksort` routine.

Draw the computation tree showing the subarrays that are recursively operated on when the `Quicksort` routine is called with the array $[4, 10, 2, 1, 20, 3, 40, 30]$.

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

Identify the problem encountered in this case. Make changes to the `Quicksort` routine to eliminate the problem, while still using the `Partition` routine to perform partitioning based on the mean of the given array. For your convenience, the `Quicksort` routine is reproduced here.

```
Quicksort(r[], lb, ub):
  if lb ≥ ub {
    return;
  }
  pivot_idx ← Partition(r[], lb, ub);
  Quicksort(r[],            ,            ); /* fill in the parameters */
  Quicksort(r[],            ,            ); /* fill in the parameters */
```

3. (20 points) Consider the Kruskal's algorithm for the construction of a minimum spanning tree:

```
 Kruskal-MST(G, w):
01.    A ← ∅;
02.    for each edge e in E[G] {
03.      Enqueue(PQ, e); // put edges in a priority queue
04.    }
05.    while A is not a spanning tree {
06.      e ← Dequeue(PQ);
07.      if {e} ∪ A does not create a cycle {
08.        A ← A ∪ {e};
09.      }
10.    }
11.    return A
```

Consider the use of a *min-heap* (**not** a max-heap) 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. To *enqueue* an edge into the priority queue, the edge is inserted at the end of the array and you use a routine similar to `Upward_Heapify` to maintain the min-heap property. For your convenience, the routine `Upward_Heapify` for inserting an integer at the end of the array for a *max-heap* of integers is given below.

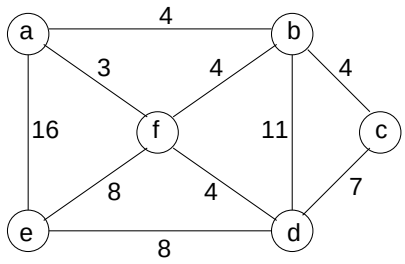
```
Upward_heapify(r[], n):
   new ← r[n-1];
   child ← n-1;
   parent ← (child-1)/2;
   while r[parent] < new and child > 0 {
     r[child] ← r[parent];
     child ← parent;
     parent ← (child-1)/2;
   }
   r[child] ← new;
```

```
Downward_heapify(A[], i, n):
   temp_r ← A[i-1];
   while i ≤ n/2 {
     j ← 2 × i;
     if j < n and A[j-1] < A[j] {
       j ← j+1;
     }
     if temp_r ≥ A[j-1] {
       break;
     } else {
       A[i-1] ← A[j-1];
       i ← j;
     }
   }
   A[i-1] ← temp_r;
```

In lines 05–10 of `Kruskal-MST`, the edges are processed for the construction of a minimum spanning tree. To *dequeue* an edge from the min-heap, the edge at the first location of the array is removed (and returned later as the dequeued item). To maintain the min-heap property, the last item in the array is moved to the vacated location, and a routine that is similar to `Downward_Heapify` is used. The size of the heap is also decremented. For your convenience, the routine `Downward_Heapify` for maintaining a max-heap of integers is given above.

This page contains only information and no questions.

The `Kruskal-MST` function is applied to the following graph:



(a) (10 points) Assume that the edges are enqueued in the following lexicographical order (first node is the primary key and the second node is the secondary key):

(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 *min-heap* data structure after the complete execution of the for-loop in lines 02–04 of `Kruskal-MST`.

(b) (10 points) Write down the edges in the order in which they are processed in lines 05–10 of `Kruskal-MST`. Identify the edges that are included in the minimum spanning tree constructed by `Kruskal-MST`.

Question	Max	Score
1	20	
2	20	
3	20	
Total	60	