ECE368 Exam 2
Fall 2013

Tuesday, Nov 12, 2013
6:30-7:30pm
FRNY G140

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 2 or question 3. Learning objective 3 can be achieved by scoring 10 points or more in question 1 or question 2. Learning objective 4 can be satisfied if you score 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 exercise and will be subject to possible disciplinary action."

`Printed Name:`

`login:`

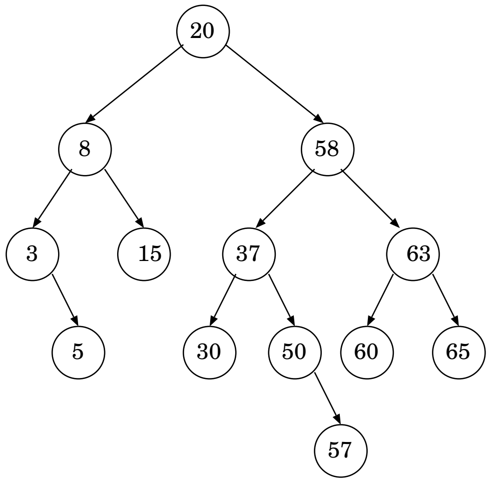
`Signature:`

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

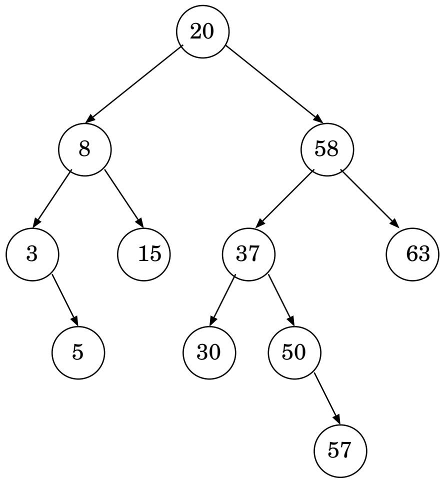
DO NOT BEGIN UNTIL INSTRUCTED TO DO SO ...

1. (20 points) Height-Balanced Binary Search Trees.

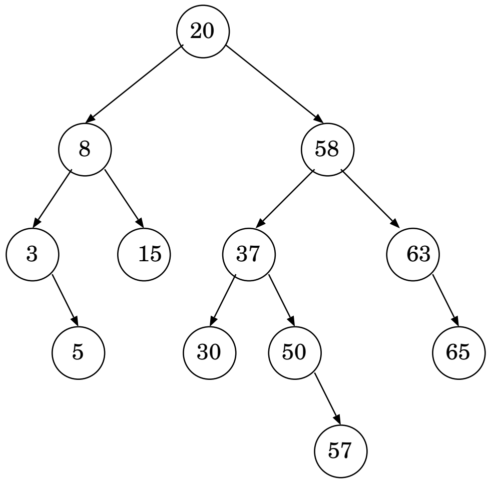
(a) (5 points) The following figure shows the binary search tree obtained right after the insertion of a new node into a height-balanced binary search tree. For this insertion, no rotations are required to balance the tree. There are a few nodes in the tree that could be the newly inserted node. Identify *all* of them. Node 5, for example, is not one of them.

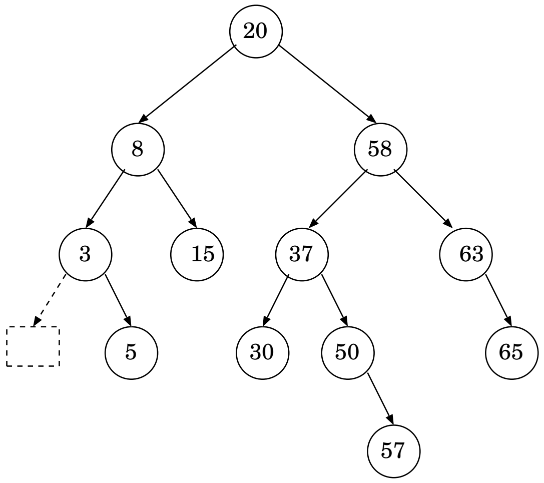


(b) (5 points) The following figure shows a binary search tree that has become *unbalanced* after the insertion of *a* new node. Identify the newly inserted node and perform rotation(s) to balance the tree.

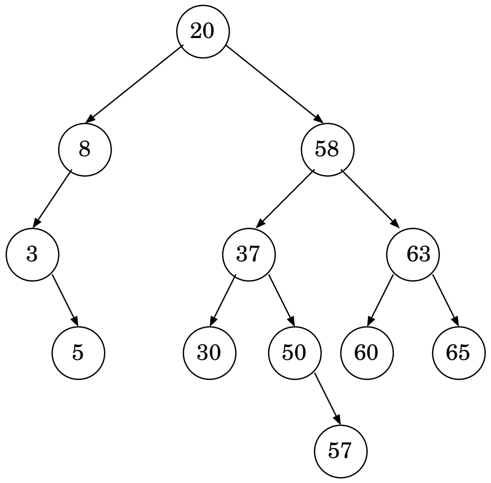


(c) (5 points) The figure on the left shows the binary search tree obtained right after the deletion of a leaf node from a height-balanced binary search tree. For this deletion, no rotations are required to balance the tree. The deleted leaf node could be at one of several locations of the original height-balanced binary search tree. Identify *all* possible locations of the deleted leaf node in the figure. One of the possible locations has been identified in the figure on the right.





(d) (5 points) The following figure shows a binary search tree that has become *unbalanced* after the deletion of a key in a non-leaf node. Identify *all* possible locations of the non-leaf node and perform rotation(s) to balance the tree.

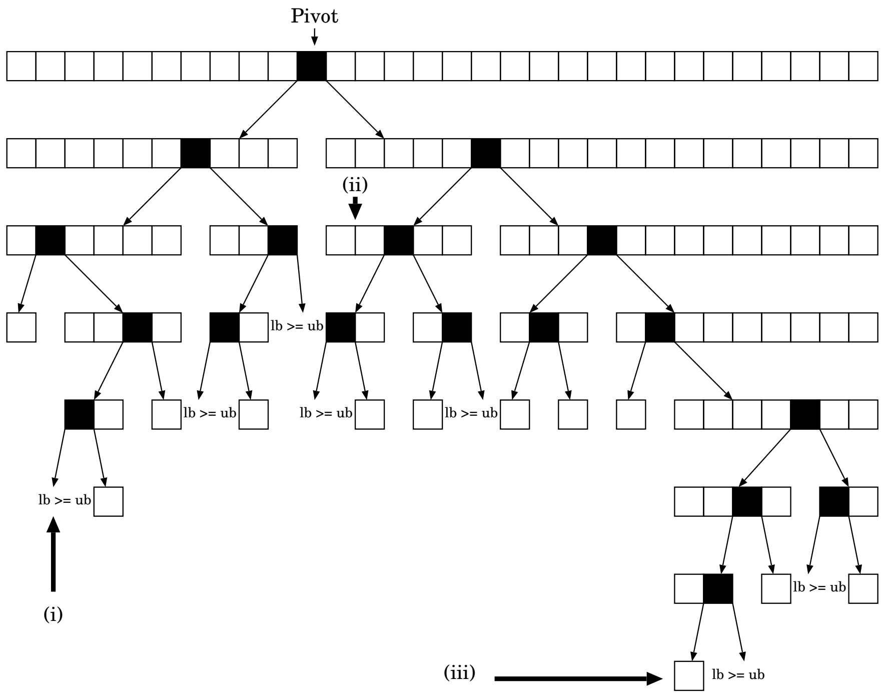


2. (20 points) Sorting.
(a) (6 points) Quicksort Using Recursions. In the class, we perform partitioning and quicksort as follows:

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

The following figure shows the computation tree of the function `Quicksort` on an array of 30 integers, with each square representing an integer in the array. The filled squares in the figure are the pivots used and placed at the right locations by the function `Partition`.

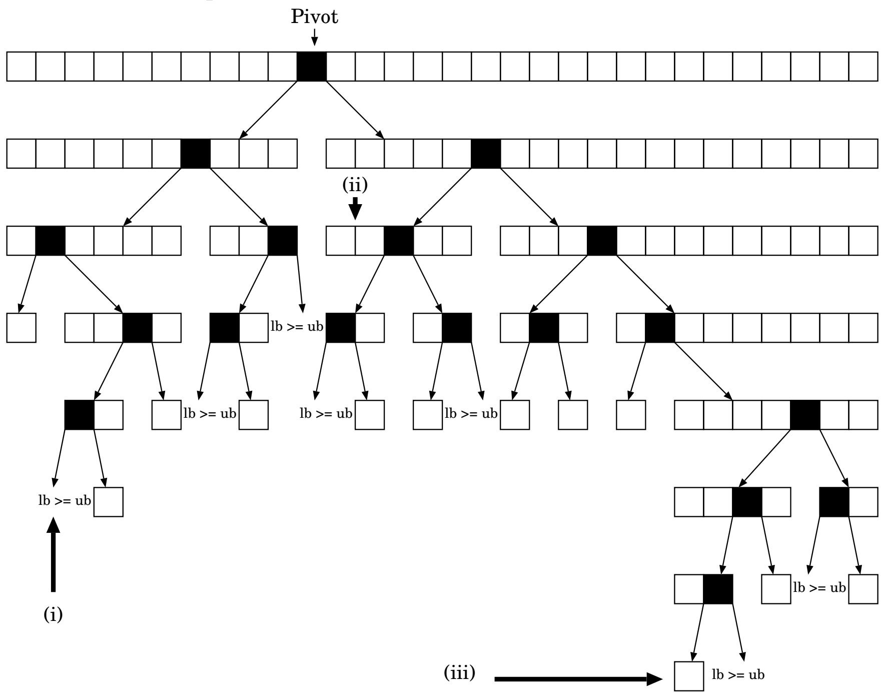


Write down the number of stack frames corresponding to the function `Quicksort` when the computation reaches (i), (ii), and (iii) in the computation tree.

(b) (14 points) Quicksort with Tail Recursion Removed. The following pseudo-code, also covered in class, removes the tail recursion in the Quicksort algorithm.

```
Quicksort_tr(r[], lb, ub):
  while (lb < ub) {
    pivot_idx ← Partition(r[], lb, ub);
    if (pivot_idx < (lb+ub)/2) {
      Quicksort_tr(r, lb, pivot_idx-1);
      lb ← pivot_idx+1;
    } else {
      Quicksort_tr(r, pivot_idx+1, ub);
      ub ← pivot_idx-1;
    }
  }
```

(4 points) For the same array of 30 integers in question **2(a)**, indicate in the following figure, which part of the computation is carried out as an iteration and which part is carried out as a recursion (recall the in-class quiz). Treat the first call of the function `Quicksort_tr` as a recursion.



(6 points) Write down the number of stack frames corresponding to the function `Quicksort_tr` when the computation reaches (i), (ii), and (iii) in the computation tree.

(4 points) Also identify the location(s) in the computation tree when the call stack has the most number of stack frames corresponding to the function `Quicksort_tr`. At any of these locations, how many stack frames corresponding to the function `Quicksort_tr` are on the call stack?

3. (20 points) Graphs.

(a) (5 points) Adjacency List Representation. For the following graph, draw the adjacency list representation of the graph. The array of vertices (`V[G]`) (shown to the right of the graph) and the adjacency lists of vertices (`adj[u]`) should all be in alphabetical order.

$$\begin{tikzcd}
m \arrow[dr] \arrow[dddr] & & n \arrow[rr] \arrow[dl] \arrow[dd] & & o \arrow[dr] \arrow[dd] & & p \arrow[ll] \arrow[dl] \arrow[dddl] \\
& q \arrow[dl] & & r \arrow[dl] \arrow[dd] & & s \arrow[ll] & \\
t & & u \arrow[ll] & & v \arrow[rr] & & w \arrow[dl] \\
& x \arrow[urrr] & & y \arrow[ur] & & z &
\end{tikzcd}$$

(b) (5 points) Topological Sort. For the same directed acyclic graph in **3(a)**, show the ordering of vertices produced when you run depth-first-search-based topological sort. The depth-first-search algorithm is given below.

```
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
```

(c) (10 points) Minimum-Cost Spanning Trees (MST). Consider the Prim's algorithm for the computation of a minimum-cost spanning tree for an undirected graph $G(V,E)$.

```
Prim(G, w, s):
  PQ ← V[G];
  for each node u in PQ {
    k[u] ← ∞;
  }
  k[s] ← 0;
  parent[s] ← NULL;
  while PQ not empty {
    u ← Extract_Min(PQ);
    for each node v in adj[u] {
      if v in PQ and w(u,v) < k[v] {
        k[v] ← w(u,v);
        parent[v] ← u;
      }
    }
  }
```

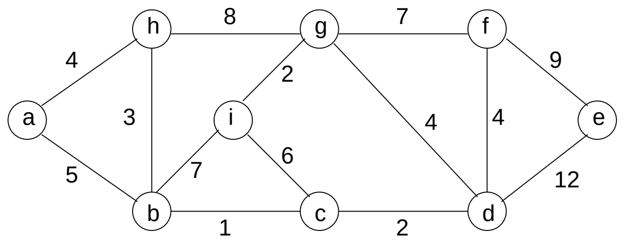
Here, a min-heap is used to implement the priority queue `PQ`. In such an implementation, you use an array to store the weighted edges and also a field to indicate the size of the heap. You use the functions `Upward_heapify` and `Downward_heapify` to maintain the priority queue. In the case of maintaining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

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

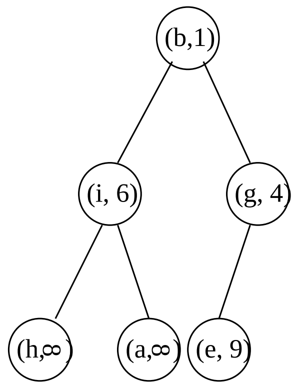
```
Downward_heapify(r[], i, ub):
  temp_r ← r[i];
  while (2*i+1 <= ub) {
    j ← 2*i + 1;
    if j < ub and r[j] < r[j+1] {
      j ← j+1;
    }
    if temp_r ≥ r[j] {
      break;
    } else {
      r[i] ← r[j];
      i ← j;
    }
  }
  r[i] ← temp_r;
```

This page contains only information and no questions.

(i) (6 points) The `Prim` function is applied to the following undirected graph:



After executing the `Prim` function for a while, the priority queue *PQ* of `k[·]` associated with the vertices (shown in parentheses in each circle) looks as follows in the form of a min-heap:



Show the new min-heap after executing one more iteration of the following instructions:

```
u ← Extract_Min(PQ);
for each node v in adj[u] {
  if v in PQ and w(u,v) < k[v] {
    k[v] ← w(u,v);
    parent[v] ← u;
  }
}
```

Assume that the adjacency lists are ordered alphabetically.

(ii) (4 points) Let the degree of node `u` in $G(V,E)$ be d_u. Write down the worst-case time complexity for executing one iteration of the following instructions:

- `u ← Extract_Min(PQ);`

-
  ```
  for each node v in adj[u] {
      if v in PQ and w(u,v) < k[v] {
        k[v] ← w(u,v);
        parent[v] ← u;
      }
  }
  ```