ECE368 Exam 2
Spring 2010

Wednesday, April 14, 2010
6:30-7:30pm
HORT 117

READ THIS BEFORE YOU BEGIN

This is a *closed book* exam. Calculators are not needed but they are allowed. Since time allotted for this exam is *exactly* 60 minutes, you should work as quickly and efficiently as possible. *Note the allocation of points and do 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* for your work.

This exam consists of 9 pages (including this cover sheet and a grading summary sheet at the end); 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."

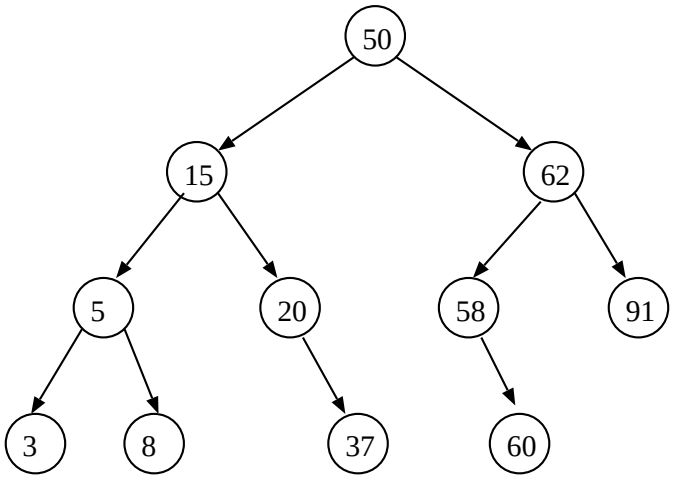
`Printed Name:`

`login:`

`Signature:`

DO NOT BEGIN UNTIL INSTRUCTED TO DO SO ...

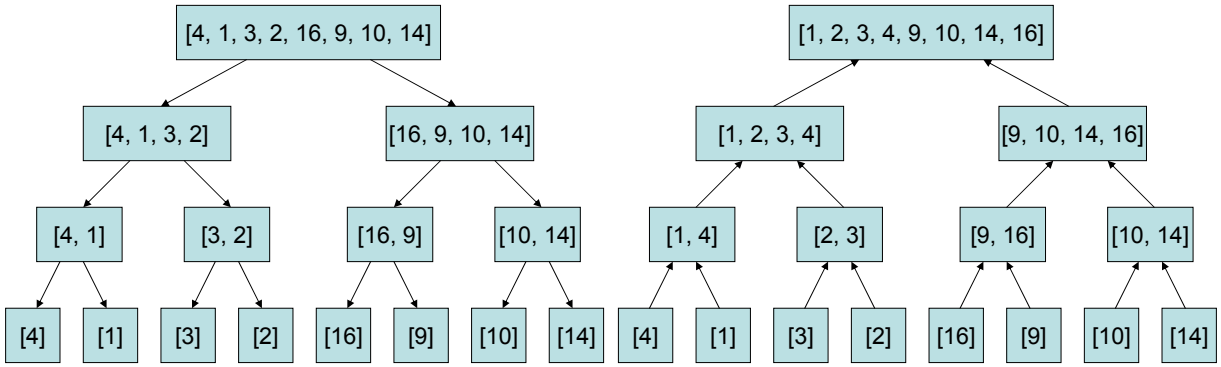
1. (30 points) The following diagram shows a *height-balanced* binary search tree (AVL-tree). Performs the following successive operations: Insert 25, insert 19, and delete 62. When you delete a non-leaf node with two child nodes, replace the deleted node with its immediate in-order successor. Draw the resulting height-balanced binary search tree after each insertion or deletion (*not after each rotation*). Clearly state the rotation operations that you have used. Use the blank page opposite this page if necessary.



2. (30 points) a. (10 points) The following routines perform an even partitioning of an array, followed by merging, i.e., the array is divided into two subarrays, and `Mergesort` is recursively applied on the two subarrays. Note that this is the `Mergesort` algorithm introduced in class.

```
Mergesort(r[], lb, ub):  // assume auxiliary space tmp[]
   if lb ≥ ub
     return
   mid ← (lb+ub)/2
   Mergesort(r, lb, mid)
   Mergesort(r, mid+1, ub)
   Merge(r, lb, mid, ub)

Merge(r[], lb, mid, ub):  // assume auxiliary space tmp[]
   memcpy(&tmp[lb], &r[lb], mid-lb+1)
   memcpy(&tmp[mid+1], &r[mid+1], ub-mid)
   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++]
   }
```

The left figure shows the partitioning process and the right shows the merging process when `Mergesort` is applied to $r = [4,1,3,2,16,9,10,14]$.

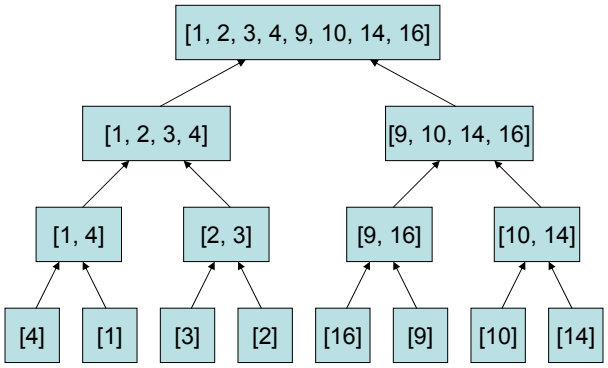


Consider an unsorted array $r = [4,1,3,2,16,9,10,14,8,7]$. As in the preceding figures, show the partitioning process and merging process when `Mergesort` is applied to $r[0..9]$. Use the opposite blank page.

b. (10 points) Mergesort can also be implemented iteratively. The following routine, called `Iter-Mergesort`, sorts an array $r[0..n-1]$ of n integers:

```
Iter-Mergesort(r[], n):  // assume auxiliary space tmp[]
  size ← 1
  while size < n {
    i ← 0
    while i < n - size {
      Merge(r, i, i+size-1, min(i+2*size-1, n-1))
      i ← i + 2*size
    }
    size ← 2*size
  }
```

The following figure shows the application of `Iter-Mergesort` on $r = [4,1,3,2,16,9,10,14]$.



Consider an unsorted array $r = [4,1,3,2,16,9,10,14,8,7]$. As in the preceding figure, show the application of `Iter-Mergesort` on $r[0..9]$.

c. (10 points) If we assume that the array `tmp[]` already stores the sorted subarrays, we can eliminate the copying of data from `r[]` to `tmp[]` and re-write the `Merge` routine as follows:

```
New-Merge(r[], tmp[], lb, mid, ub):  // assume tmp[] stores sorted subarrays
  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++]
  }
```

The following routine is an attempt to use the `New-Merge` routine:

```
Iter-Mergesort(r[], n):  // assume auxiliary space tmp[]
  sorted ← r; // r[] contains sorted 1-element subarrays
  merged ← tmp; // tmp[] to store the merged results
  size ← 1
  while size < n {
    i ← 0
    while i < n - size {
      New-Merge(merged, sorted, i, i+size-1, min(i+2*size-1, n-1))
      i ← i + 2*size
    }
    sorted ↔ merged // exchange roles
    size ← 2*size
  }
  if sorted == tmp // tmp contains sorted array
    memcpy(r, tmp, n)
```

Does the above routine work properly? Justify your answer. If it does not work, provide a remedy.

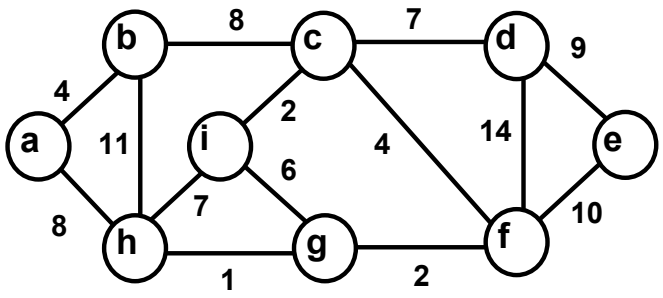
3. (40 points) Consider the following pseudo code for a disjoint-set implementation.

```
Make-Set(x):
  p[x] ← x
  rank[x] ← 0

Find-Set(x):
  if x ≠ p[x]
    return Find-Set(p[x])
  return p[x]

Link(x, y):
  if rank[x] > rank[y]
    p[y] ← x
  else {
    p[x] ← y
    if rank[x] == rank[y]
      rank[y] ← rank[y] + 1
  }
```

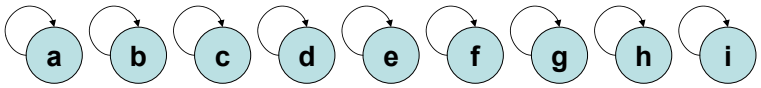
Now, consider the Kruskal's algorithm for the construction of a minimum spanning tree using the disjoint-set data structure:

```
Kruskal-MST(G, w):
  A ← ∅
  for each node u in V[G]
    Make-Set(u)
  for each edge e = (u,v) in E[G]
    Enqueue(PQ, e) // put edges in a priority queue
  while A is not a spanning tree {
    e = (u,v) ← Dequeue(PQ)
    if ((x ← Find-Set(u)) ≠ (y ← Find-Set(v)) {
      A ← A ∪ {e = (u,v)}
      Link(x,y)
    }
  }
  return A
```

Consider the following graph:



The disjoint-set data structure constructed by the for-loop in the `Kruskal-MST` routine is as follows, with each `p[x]` field depicted by an arrow:



If we perform `Dequeue(PQ)` until the priority queue `PQ` is empty, the edges are dequeued in the following order: (g,h), (c,i), (f,g), (a,b), (c,f), (g,i), (c,d), (h,i), (a,h), (b,c), (d,e), (e,f), (b,h), and (d,f).

(a) (10 points) Show the disjoint-set data structure that results after processing the edges (g,h), (c,i), (f,g), (a,b), and (c,f) with the `Kruskal-MST` routine.

(b) (10 points) Show the disjoint-set data structure that results after the following edges are processed by the `Kruskal-MST` routine: (g,i), (c,d), (h,i), (a,h), and (b,c).

(c) (10 points) Show the disjoint-set data structure when the `Kruskal-MST` routine terminates.

(d) (10 points) How do you determine whether `A` in the `Kruskal-MST` routine is a spanning tree? Describe two different approaches that allow you to evaluate in $O(1)$ time complexity the condition in the loop "`while A is not a spanning tree`" in each iteration.

Question	Max	Score
1	30	
2	30	
3	40	
Total	100	