ECE368 Exam 2
Spring 2017

Wednesday, April 12, 2017
6:30-7:30pm
CL50 224

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, that I have not copied the work of any other student while completing it, that I have not obtained help from any other student, and that I will not provide help to any other student. 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 disciplinary action as outlined in the course policy."

`Printed Name:`

`Signature:`

If the statement is not signed, the exam will not be graded and it will not be returned.

This is an *open-book, open-notes* exam. Electronic devices are not allowed.

Be *concise*. When you are asked of the time complexity of an algorithm, it is *not* necessary to do a line-by-line analysis of the algorithm. *A general explanation would suffice*.

Assume that all necessary "`.h`" files are included and all `malloc` function calls are successful.

This exam consists of 8 pages; it is *your responsibility* to make sure that you turn in a complete copy of the exam.

If you are not clear about what a question is asking, you can explain what you are answering (e.g., "I think this question is asking for ...") or you can state assumptions that you are making (e.g., "I assume that the entry -1 is equivalent to NULL").

If you score 50% or more for a question, you are considered to have met all learning objectives associated with that question.

Your Purdue ID should be visible for us to verify your identity.

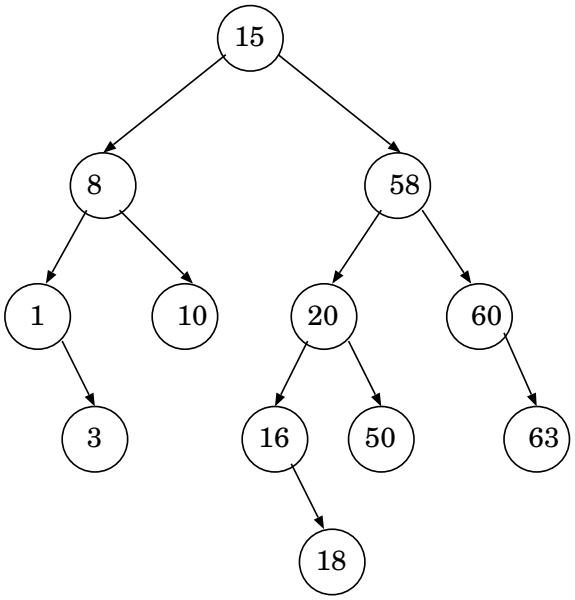
If you finish the test before 7:25pm, you may turn in the test and leave. **After 7:25pm, you have to stay until we release the entire class. Stop writing at 7:30pm. If you continue to write, that is considered cheating.**

When we collect the copies of test, you are to remain in your seat in an orderly manner until we have collected and counted all copies.

DO NOT BEGIN UNTIL INSTRUCTED TO DO SO ...

1. Height-Balanced Binary Search Trees (12 points) (Learning Objective 1)

You are given the following height-balanced binary search tree:



For each of the following insertion and deletion operations (in **questions 1(a)–(c)**), you perform the operation on the preceding height-balanced binary search tree. **Do not apply the operations successively.**

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

If a deletion operation 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.

(a) (4 points) Insert 61.

(b) (4 points) Delete 60.

(c) (4 points) Delete 15.

2. Mergesort (16 points) (Learning Objectives 2 and 3)

The function `Iterative_mergesort` performs merge sort iteratively (**as covered in class**).

```
1  void Iterative_mergesort(int *r, int size)
2  {
3     int subarray_size = 1;
4     while (subarray_size < size) { // there are subarrays
5        int i = 0; // index of left subarray
6        while (i + subarray_size < size) {  // right subarray exists
7           Merge(r, i, i + subarray_size - 1,
8                    i + 2*subarray_size - 1 < size?
9                    i + 2*subarray_size - 1: size - 1);
10          i += 2*subarray_size; // index of next left subarray
11       }
12       subarray_size *= 2; // double the size of subarrays
13    }
14 }
```

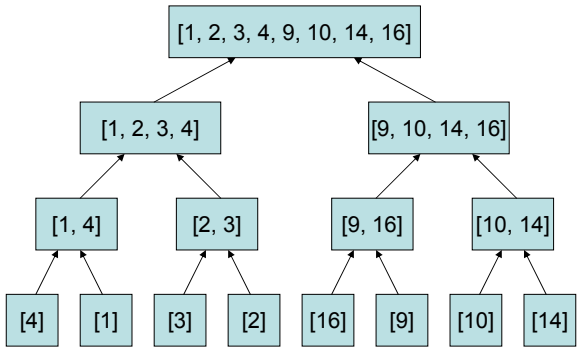
It calls the function `Merge` to merge two sorted subarrays using an auxiliary array `tmp` (**as covered in class**). **You may assume that the auxiliary array is a global variable that has been appropriately allocated.**

```
15 void Merge(int *r, int lb, int mid, int ub)
16 {
17    int m;
18    for (m = lb; m <= ub; m++) { tmp[m] = r[m]; } // copy to tmp
19    int i = lb;
20    int j = mid+1;
21    for (m = lb; m <= ub; m++) { // merge from tmp to r
22       if (i > mid) { r[m] = tmp[j++]; }
23       else if (j > ub) { r[m] = tmp[i++]; }
```

```
24          else if (tmp[j] < tmp[i]) { r[m] = tmp[j++]; }
25          else { r[m] = tmp[i++]; }
26       }
27    }
```

The figure to the right shows the application of `Iterative_mergesort` by calling in the `main` function:

```
int r[] = {4, 1, 3, 2, 16, 9, 10, 14};
Iterative_mergesort(r, 8);
```



Now, you are given a different merge sort algorithm. **The two functions below are new.** The `Natural_mergesort` function iteratively looks for non-descending subarrays within the given array `r` and merges these non-descending subarrays using the `Merge` function (Lines 15–27). The `Natural_mergesort` function calls the function `find_end_of_run` to find the index of the last item in a non-descending subarray.

```
28 void Natural_mergesort(int *r, int size)
29 {
30    int number_of_runs; // number of non-descending subarrays
31    do {
32       number_of_runs = 0; // no non-descending subarrays initially
33
34       int i = 0; // index of left subarray
35       while (i < size) { // left subarray exists
36          number_of_runs++; // found one non-descending subarrays
37
38          // find end of left non-descending subarray
39          int end_of_left_run = find_end_of_run(r, size, i);
40
41          // check whether a right subarray exists
42          if (end_of_left_run + 1 < size) { // right subarray exists
43             // find end of right non-descending subarray
44             int end_of_right_run =
45                find_end_of_run(r, size, end_of_left_run+1);
46             Merge(r, i, end_of_left_run, end_of_right_run);
47             i = end_of_right_run + 1; // index of next left subarray
48          } else {  // absence of right subarray
49             break; // break from inner while-loop
50          }
51       } // while left subarray exists
52    } while (number_of_runs > 1); // do while two or more subarrays
53 }
```

```
54 int find_end_of_run(int *array, int size, int start_of_run)
55 {
56    int end_of_run = start_of_run; // assume a single-integer subarray
57    // look for a smaller item to end the non-descending subarray
58    while ((end_of_run+1 < size)
59           && (array[end_of_run] <= array[end_of_run+1])) {
60       end_of_run++;
61    }
62    return end_of_run;
63 }
```

In **questions 2(a)–(c)**, consider the application of `Natural_mergesort` as follows:

```
int r[] = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7};
Natural_mergesort(r, 10);
```

(a) (4 points) Enter in the following table the corresponding values of `i`, `end_of_left_run`, and `end_of_right_run` calculated in the **first iteration** of the `do-while` loop (Lines 31–52). If there are no corresponding values of `end_of_left_run` and/or `end_of_right_run`, put "N.A." in the corresponding table entries. There may be more columns than necessary in the table.

i	0				
end_of_left_run					
end_of_right_run					

(b) (4 points) Draw a figure to demonstrate the merging of non-descending subarrays performed in the **first iteration** of the `do-while` loop. Clearly indicate the subarrays that are being merged and the results of the merging operations.

(c) (4 points) Draw a figure to demonstrate the merging operations performed in the **remaining iterations** of the `do-while` loop. Clearly indicate the subarrays that are being merged and the results of the merging operations.

(d) (4 points) When `Natural_mergesort` is applied to an array `r` containing n integers, what is the best-case time-complexity in terms of n? What is the worst-case time-complexity in terms of n. Justify your answers.

Best-case time-complexity:

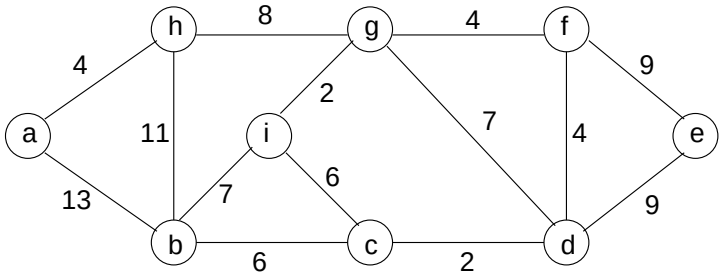
Worst-case time-complexity:

3. Minimum Spanning Tree (12 points) (Learning Objectives 1, 3, and 4)

The Kruskal's algorithm below (**as covered in class**) constructs a minimum spanning tree:

```
 Kruskal-MST(G, w):
01.    A ← ∅;
02.    Build a priority queue PQ of edges in E[G] using min-heap;
03.    for each node u in V[G]
04.      Make-Set(u);
05.    while A is not a spanning tree {
06.      e = (u,v) ← Dequeue(PQ);
07.      if ((x ← Find-Set(u)) ≠ (y ← Find-Set(v))) {
08.        A ← A ∪ {e = (u,v)};
09.        Link(x,y);
10.      }
11.    }
12.    return A;
```

(a) (7 points) The `Kruskal-MST` function is applied to this undirected graph $G(V,E)$ in **questions 3(a)–(b)**:



A *min-heap* is used to implement the priority queue `PQ`. **In the class**, we build a *max-heap* of n integers stored in array `r[0..n-1]` as follows:

```
// 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;
```

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

(a,b)	(a,h)	(b,c)	(b,h)	(b,i)	(c,d)	(c,i)	(d,e)	(d,f)	(d,g)	(e,f)	(f,g)	(g,h)	(g,i)

Show the ordering of the edges (*not* their weights) in the array of the *min-heap* data structure constructed in Line 02 of `Kruskal-MST` using an approach similar to that for the construction of a max-heap of integers with `Downward_heapify`.

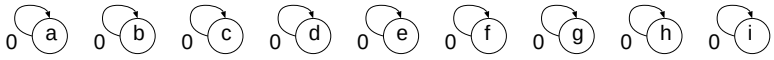
(b) (5 points) Consider the following pseudo code for a disjoint-set implementation (**as covered in class**).

```
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;
  }
```

The disjoint-set data structure constructed by the for-loop in Lines 03–04 of the `Kruskal-MST` routine is as follows, with each `p[x]` field depicted by an arrow. The integer beside each node is the rank.



Show the disjoint-set data structure after the following edges have been dequeued and processed (Lines 05–11): (c,d), (g,i), (a,h), (d,f), (g,f), (b,c), (c,i). (Note that only these edges do not form the complete set of of edges and the ordering given is not related to the min-heap you have constructed in **3(a)**.) For your convenience, the graph is reproduced here: