COMPUTER SCIENCE 112 - Spring 2013
FINAL EXAM

Name: _____

CIRCLE your section: W 10:35 W 12:15 W 1:55 Th 3:35 Th 5:15 Th 6:55

- Be sure your exam has 7 questions.
- **DO NOT TEAR OFF THE SCRATCH PAGES OR REMOVE THE STAPLE.**
- Be sure to fill your name and circle your recitation time above, and your name in all subsequent pages where indicated.
- This is a CLOSED TEXT and CLOSED NOTES exam. You MAY NOT use calculators, cellphones, or any other electronic device during the exam.
- In all questions involving analysis, think carefully and write your answer clearly and precisely, including all the steps that are needed to get the answer. Credit will not be given if your answer comes without a proper sequence of steps with clear explanation.
- NOTE!!! You will only get partial credit for any answer if it is (a) clearly legible and coherent, and (b) close enough to the correct solution that we can infer you know the material reasonably well. A lack of understanding of the concept involved will get little or not credit, even if you have plowed through some mechanics.

Do not write below this line

Problem	Value	Score
1 Quicksort	20	_____
2 Shortest path	20	_____
3 Heapsort	20	_____
4 Dijkstra's Algorithm	20	_____
5 Heap Merge	20	_____
6 Hash Table	20	_____
7 Word Count	20	_____
TOTAL	140	_____

1. **Quicksort (20 pts;10+10)**

(a) Given the following array of integers:

```
24    6    72    28    12    9    3
```

Execute quicksort on this array (use quicksort all the way through) using the first entry as the pivot. Show the resulting quicksort recursion tree, in which the root of each (sub) tree is the (sub) array to be sorted recursively.

(b) Derive the *best case* big O running time for quicksort, counting only pair-wise comparisons between items in the array, one unit of time per comparison. Show clearly all the steps in how these comparisons add up to the final big O time. If you only give the answer without showing all the work with clear explanations, you will **not** get any credit.

2. **Shortest Path (20 pts)**

The following is an implementation of an *undirected* graph without any weights on the edges. Assume that the graph has already been loaded into the adjacency linked lists. Fill in the required method to find the shortest path from vertex numbered `x` to vertex numbered `y`. (Note: Since the graph is unweighted, the length of a path is simply the number of edges in it.) You may use, without implementation, any of the standard methods of the `Queue` and `Stack` classes. For anything else, you will need to implement your own code (including helper methods, if needed). You may add fields to the `Graph` class as necessary.

```
public class Neighbor {
   public int vnum;
   public Neighbor next;
   ...
}

public class Graph {
   Neighbor[] adjLists;  // adjacency linked lists for all vertices
   // add other fields as necessary









   // returns the length of the shortest path from x to y, (assume y different from x)
   // or -1 if y is not reachable from x
   public int shortestPath(int x, int y) {
     // FILL IN THIS METHOD
```

198:112 Spring 2013 Final Exam; Name: _____

3. **Heapsort (20 pts;11+4+5)**

You are given the following `Heapsort` class outline - fill in the `siftDown`, `buildHeap`, and `sortHeap` methods.

```
public class Heapsort {

   // sorts an array in ascending order using heapsort
   public static <T extends Comparable<T>>
   void sort(T[] list) {
      buildHeap(list);
      sortHeap(list);
   }

   // sifts down in an array[0..n-1] starting at index k
   private static <T extends Comparable<T>>
   void siftDown(T[] list, int k, int n) {
       // FILL IN THIS METHOD









   }
```

198:112 Spring 2013 Final Exam; Name: _____

```
    // arranges items in list in heap order in LINEAR TIME, using repeated sift downs
    private static <T extends Comparable<T>>
    void buildHeap(T[] list) {
        // FILL IN THIS METHOD









    }

    // sorts a heap-ordered array
    private static <T extends Comparable<T>>
    void sortHeap(T[] list) {
        // FILL IN THIS METHOD









    }
}
```

4. **Dijkstra's Algorithm (20 pts;6+6+4+4)**

Dijkstra's single source shortest paths algorithm finds shortest paths from a (source) vertex to all other vertices. Once the algorithm is executed on a source, say *x*, the distances to all vertices from this source can be stored in a file. Subsequently, to determine the shortest distance from *x* to any vertex, the algorithm need not be executed again; instead, the distance is simply read off the file.

Consider an application that works on a directed graph that has 1,024 vertices and 10,240 edges, and positive edge weights. (**Remember,** $log_2(1,024)$ **= 10**)

- In any single run, the application gets the shortest distance from *x* to *y* in the graph, where *x* and *y* are input vertex numbers.
- It maintains a single file into which it can dump shortest distances from any vertex (so at any time this file can contain sets of shortest distances from numerous vertices)
- It takes exactly **(n+e)*log(n)** time to execute Dijkstra's algorithm for a given source vertex (This is not a big-oh order; it is an exact number.)
- It takes exactly **100*log(n)** time to find whether the shortest distance from *x* to *y* is in the distance file, and if so, the actual distance. Let's call this process "file lookup".
- It takes exactly **100*log(n)** time to store in the file the shortest distances from any source x - call this process "file store".

a) What would your strategy be to determine the shortest distance from *x* to *y*, for any single run of the application with x and y as parameters? Write your strategy in the form of an algorithm, i.e. a sequence of steps. **Do not write any Java code.**

b) Given your strategy in (a), what would be the **average** time to find the shortest distance from *x* to *y*, for the first 100 *(x,y)* inputs? Assume that the application starts with an empty file, and that each distinct *x* appears exactly five times in these 100 *(x,y)* pairs. Show your work.

c) If, instead of **100*log(n)**, it took **k*log(n)** time for file lookup of distance, how would your strategy of (a) depend on the value of *k*, for a graph with 1,024 vertices and 10,240 edges? Explain.

d) If we fixed the file lookup time at **100*log(n)**, then how would your strategy of (a) depend on general values of n and e? Explain.

5. **Heap Merge (20 pts;5+5+3+7)**

Suppose that we have k heaps, with n elements in each. We wish to combine these into a single heap. Each of the following sub-parts describes one approach to merging the heaps, and you are asked to derive the big O running time. To get any credit, **you must show your work**.

a) We create a new, empty heap H. For each of the k heaps, we repeatedly remove the highest priority element and insert it into H, until the original heap is empty. What is the worst-case big O running time?

b) We create a new, empty linked list L. For each of the k heaps, we repeatedly remove the highest priority element and insert it at the front of L, until h_i is empty. We then transfer the elements of L into an array, and **heapify** the array (i.e. arrange in heap order) using the fastest known algorithm. What is the worst-case big O running time?

c) We create a new, empty linked list L. For each of the k heaps, we iterate over its array storage, remove each element, and add it to the front of L. We then transfer the elements of L into an array, and **heapify** the array using the fastest known algorithm. What is the worst-case big O running time?

d) We group the k heaps into $k/2$ pairs, and apply the algorithm from part (c) on each pair, leaving $k/2$ heaps. We then repeat this process until we are left with a single heap. What is the worst-case big O running time?

6. **Hash Table (20 pts;6+7+7)**

a) You are given a hash table of initial capacity 5, and need to use the hash function **h(key) = key mod table_capacity**. Create the smallest dataset of distinct non-negative integer keys that will be inserted in the hash table so that, after inserting all the integers in the dataset, the average number of comparisons for successful search is *less than* 3, and the worst case number of comparisons is 6. Show the hashtable with all your dataset integers in it, and work out the average. (Assume that the load factor threshold is large enough that no rehashing will happen, and that all items are equally likely to be searched.)

b) Create the smallest dataset for the same requirements as above, plus an additional constraint that the load factor threshold is 2, and you must force exactly one rehash. Assume that a rehash doubles the capacity of the table. Show the contents of the hashtable just before rehash, as well as the final hashtable, and work out the average.

c) Suppose you insert 125 integer keys into a hash table with an initial capacity of 25 and a load factor threshold of 2. Assume the hash **h(key) = key mod table_capacity** is used. How many total units of work will be done by the time all keys are inserted if it takes one unit of work to compute $h(key)$ and one unit of work to insert an entry into a linked list? (Count ONLY these.) Assume a rehash doubles the table capacity, and the load factor is compared against the threshold *before* an item is inserted.

7. **Word Count (20 pts;6+7+7)**

You have been asked to write a program to count the number of times each word occurs in a text file, and print them grouped according to word length. Below is a sample text file on the left, and the expected output of your program on the right:

```
This is the first line,                    is: 2
and this line is the second line.          the: 2
                                           and: 1
                                           this: 2
                                           line: 3
                                           first: 1
                                           second: 1
```

Note that words are counted ignoring case, and are printed in order of word length. In any group of words of the same length, the printout can be in any order (e.g. *the* and *and* are both printed before words of greater length, but they need not be in any specific relative sequence.)

You implement the program using an AVL tree to count the frequencies of words. Each node in the AVL tree has a word (with its count), and words (alphabetical) are the basis of the AVL ordering.

a) Show the final AVL tree after *all* the words from the sample input file have been inserted. (Remember that words will be inserted one at a time as they are read in, but you only need to show the final tree, not all the intermediate trees). Make sure you show balance factors at all the nodes. There is *only one correct AVL tree answer*, according to the AVL insertion algorithm you have learned in class.

b) To generalize, if there are k distinct words, and n words in all (in the sample file, $k = 7$ and $n = 12$), how much time will it take in the worst case (big O) to store all the words with counts? Count word comparisons (each is unit time) only. Ignore the time taken to read a word from input.

c) Describe the most efficient algorithm and data structures you can think of, to print the output. Derive (show your work!) the worst case time big O running time of your algorithm, if the maximum word length is m? (You will get AT MOST HALF THE CREDIT if your algorithm is not the most efficient.)

SCRATCH PAGE

198:112 Spring 2013 Final Exam; Name: _____

SCRATCH PAGE

SCRATCH PAGE