Name:_____

CSE373 Fall 2013, Final Examination
December 10, 2013

Please do not turn the page until the bell rings.

Rules:

- The exam is closed-book, closed-note, closed calculator, closed electronics.
- **Please stop promptly at 4:20.**
- There are **114 points** total, distributed **unevenly** among **10** questions (many with multiple parts):

Question	Max	Earned
1	28	
2	8	
3	11	
4	15	
5	7	
6	7	
7	12	
8	8	
9	9	
10	9	

Advice:

- Read questions carefully. Understand a question before you start writing.
- **Write down thoughts and intermediate steps so you can get partial credit. But clearly circle your final answer.**
- The questions are not necessarily in order of difficulty. **Skip around.** Make sure you get to all the problems.
- If you have questions, ask.
- Relax. You are here to learn.

Name:_____

1. (**28** points) **Don't miss part (b) of this question.**

   (a) Fill in this table with the **worst-case** asymptotic running time of each operation when using the data structure listed. Assume the following:

   - Items are *comparable* (given two items, one is less than, equal, or greater than the other) in $O(1)$ time.
   - For insertions, it is the client's responsibility not to insert an item if there is already an equal item in the data structure (so the operations do not need to check this).
   - For insertions, assume the data structure has enough room (do *not* include any resizing costs).
   - For deletions, assume we do *not* use lazy deletion.

	insert	lookup	delete	getMin	getMax
	(take an item and add it to the structure)	(take an item and return if it is in the structure)	(take an item and remove it from the structure, if present in it)	(return smallest item in the structure)	(return largest item in the structure)
sorted array					
unsorted array					
array kept organized as a min-heap					
AVL tree					
Hashtable with chaining for collision resolution					

   (b) The entries in the table above are worst-case. For each one, circle the entry if we expect an asymptotically better run-time in practice. The correct answer for this question circles two entries.

Solution:
See next page

Name:_____

(a)

	insert	lookup	delete	getMin	getMax
	(take an item and add it to the structure)	(take an item and return if it is in the structure)	(take an item and remove it from the structure, if present in it)	(return smallest item in the structure)	(return largest item in the structure)
sorted array	$O(n)$	$O(\log n)$	$O(n)$	$O(1)$	$O(1)$
unsorted array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
array kept organized as a min-heap	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$
AVL tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Hashtable with chaining for collision resolution	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

For hashtable getMin and getMax, we accepted $O(n + \text{tableSize})$ since that is "even more correct" but deducted 1 point total for $O(\text{tableSize})$.

(b) Circle these two: Hashtable lookup and Hashtable delete

Name:_____

2. (**8** points) Suppose there is a class `C` that has a constructor that takes an argument of type `A` and stores the content of the `A` object for later use by other methods. For each definition of class `A` below, answer "yes copy" if the constructor in `C` would need to make a deep copy of the argument to preserve the abstraction defined by `C` or "no copy" if it is not necessary.

(a)
```
public class B {
    public int x;
    public int y;
    public B(int _x, int _y) { x = _x; y = _y; }
}
public class A {
    public B a;
    public B b;
    public A(B _a, B _b) { a = _a; b = _b; }
}
```

(b)
```
public class B {
    public final int x;
    public final int y;
    public B(int _x, int _y) { x = _x; y = _y; }
}
public class A {
    public final B a;
    public final B b;
    public A(B _a, B _b) { a = _a; b = _b; }
}
```

(c)
```
public class B {
    public final int x;
    public final int y;
    public B(int _x, int _y) { x = _x; y = _y; }
}
public class A {
    public B a;
    public B b;
    public A(B _a, B _b) { a = _a; b = _b; }
}
```

(d)
```
public class B {
    public int x;
    public int y;
    public B(int _x, int _y) { x = _x; y = _y; }
}
public class A {
    public final B a;
    public final B b;
    public A(B _a, B _b) { a = _a; b = _b; }
}
```

Solution:
(a) yes copy; (b) no copy; (c) yes copy; (d) yes copy
Note: For (c), we need to copy, but a shallow copy suffices. We gave credit if this was explained.

Name:_____

3. (**11** points) For each of the six questions in parts (a)-(c), answer in terms of big-Oh and the number of vertices in the graph $|V|$.

   (a) Suppose a graph has no edges.
      i. What is the asymptotic space cost of storing the graph as an adjacency list?
      ii. What is the asymptotic space cost of storing the graph as an adjacency matrix?

   (b) Suppose a graph has every possible edge.
      i. What is the asymptotic space cost of storing the graph as an adjacency list?
      ii. What is the asymptotic space cost of storing the graph as an adjacency matrix?

   (c) Suppose an undirected graph has one node A that is connected to every other node and the graph has no other edges.
      i. What is the asymptotic space cost of storing the graph as an adjacency list?
      ii. What is the asymptotic space cost of storing the graph as an adjacency matrix?

   (d) Is an adjacency list faster or slower than an adjacency matrix for answering queries of the form, "is edge (u, v) in the graph"?

   (e) Is an adjacency list faster or slower than an adjacency matrix for answering queries of the form, "are there any directed edges with u as the source node"?

Solution:

(a) i. $O(|V|)$
   ii. $O(|V|^2)$

(b) i. $O(|V|^2)$
   ii. $O(|V|^2)$

(c) i. $O(|V|)$
   ii. $O(|V|^2)$

(d) slower

(e) faster

Name:_____

4. (**15** points)

   (a) Draw a weighted undirected graph with exactly 3 nodes that has exactly 0 minimum spanning trees.

   (b) Draw a weighted undirected graph with exactly 3 nodes that has exactly 1 minimum spanning tree.

   (c) Draw a weighted undirected graph with exactly 3 nodes that has exactly 2 minimum spanning trees.

   (d) Draw a weighted undirected graph with exactly 3 nodes that has exactly 3 minimum spanning trees.

   (e) Argue in roughly 2-3 English sentences that no weighted undirected graph with 3 nodes can have more than 3 minimum spanning trees.

   Solution:

   (a) Any unconnected, weighted, undirected graph

   (b) Solution needs to have either 2 or 3 non-self edges and if it has 3 non-self edges, then no two can have the same weight

   (c) Graph needs 2 non-self edges, with exactly 2 having the same weight and the third edge having a lower weight

   (d) Graph needs 3 non-self edges, all with the same weight

   (e) A 3-node graph can only have 3 spanning trees because it takes 2 edges (the number of nodes minus 1) to create a spanning tree. If the nodes are A, B, and C, the only possible spanning trees are:

   - (A,B), (A,C)
   - (B,C), (A,C)
   - (A,B), (B,C)

Name:_____

5. (**7** points) Complete this Java method so that it properly implements **insertion sort**. Make sure the result is in-place (do not use another array).

```
void insertionSort(int [] array) {
     for(int i=1; i < array.length; i++) {
          // YOUR CODE HERE









      } // end of the for-loop
}
```

Solution:

```
void insertionSort(int [] array) {
    for(int i=1; i < array.length; i++) {
        int tmp = array[i];
        int j;
        for(j=i; j > 0 && tmp < array[j-1]; j--)
            array[j] = array[j-1];
        array[j] = tmp;
    }
}
```

Name:_____

6. (**7** points)

Complete this Java method so that it properly implements **selection sort**. Make sure the result is in-place (do not use another array). Notice there are two places to add code.

```
void selectionSort(int [] array) {
    for(int i=0; i < array.length; i++) {
        int index_of_least = i;
        int j;
        for(j=i+1; j < array.length; j++) {
            // YOUR CODE HERE #1









        } // end of the inner for-loop
        // YOUR CODE HERE #2









    } // end of the outer for-loop
}
```

Solution:
Note it is also okay to shift everything to the right to make room after the loop. This is more work and unnecessary, but it keeps the sort stable and does not change the asymptotic complexity.

```
void selectionSort(int [] array) {
    for(int i=0; i < array.length; i++) {
        int index_of_least = i;
        int j;
        for(j=i+1; j < array.length; j++) {
            if(array[j] < array[index_of_least])
                index_of_least = j;
        }
        int tmp = array[i];
        array[i] = array[index_of_least];
        array[index_of_least] = tmp;
    }
}
```

Name:_____

7. (**12** points) Sorting short answer: In all questions about quick-sort, assume n is large enough that any use of a cut-off is not relevant.

   (a) What is the worst-case asymptotic running time of heap-sort?

   (b) What is the worst-case asymptotic running time of merge-sort?

   (c) What is the worst-case asymptotic running time of quick-sort?

   (d) Can heap-sort be done in-place?

   (e) Can merge-sort be done in-place?

   (f) Can quick-sort be done in-place?

   (g) Consider ***one*** item in an array that is sorted with mergesort. In asymptotic (big-Oh) terms, how many times can that ***one*** item move to a different location?

   (h) What is the asymptotic running time of quick-sort if the array is already sorted (or almost sorted) and the pivot-selection strategy picks the leftmost element in the range-to-be-sorted?

   (i) What is the asymptotic running time of quick-sort if the array is already sorted (or almost sorted) and the pivot-selection strategy picks the rightmost element in the range-to-be-sorted?

   (j) What is the asymptotic running time of quick-sort if the array is already sorted (or almost sorted) and the pivot-selection strategy picks the middle element in the range-to-be-sorted?

   (k) Before starting a comparison sort for an array with n elements, how many possible orders are there for the elements in the final result?

   (l) If at some point during a comparison sort, the algorithm has enough information to narrow the final result down to k possibilities, what is the least number of possibilities that remain after performing an additional comparison?

Solution:

(a) $O(n \log n)$
(b) $O(n \log n)$
(c) $O(n^2)$
(d) yes
(e) no
(f) yes
(g) $O(\log n)$
(h) $O(n^2)$
(i) $O(n^2)$
(j) $O(n \log n)$
(k) $n!$
(l) $k/2$

Name:_____

8. (**8** points)

Suppose we use radix sort to sort the numbers below, using a radix of 10. Show the state of the sort *after each of the first two passes*, *not* after the sorting is complete. If multiple numbers are in a bucket, put the numbers that "come first" closer to the top.

Numbers to sort (in their initial order):

13, 105, 1492, 411, 1776, 1, 6, 2014, 99, 98, 96, 11, 21

Put the result after the first pass here:

0	1	2	3	4	5	6	7	8	9

Put the result after the second pass here:

0	1	2	3	4	5	6	7	8	9

Solution:

```
After first pass:
bucket:     0     1     2     3     4     5     6     7    8    9
contents:       411  1492    13  2014    105  1776        98   99
                  1                       6
                 11                      96
                 21

After second pass:
bucket:     0     1     2     3     4     5     6     7    8     9
contents:   1   411    21                      1776       1492
          105    11                                        96
            6    13                                        98
               2014                                        99
```

Name:_____

9. (**9** points)

(a) Suppose you have a (large) linked list and you want to perform some operation `f` on every element and some operation `g` on every element. Your first version of the code first does all the `f` operations in one list traversal and then all `g` operations in a second traversal. Your second version does one traversal, doing `f` and then `g` for each element. Why might the second version be faster?

i. temporal locality
ii. spatial locality
iii. it is faster asymptotically
iv. none of the above

(b) Suppose you have a large array of n data items that are in a random order and you want to return a sample of 1% of them. Your first version of the code returns (a copy of) every 100^{th} item (elements at index 0, 100, 200, etc.). Your second version returns (a copy of) the first 1% of the items (elements at index 0 up to $n/100$). Why might the second version be faster?

i. temporal locality
ii. spatial locality
iii. it is faster asymptotically
iv. none of the above

(c) Suppose you have a large linked list of n integers and you want to print them in reverse order (the numbers closer to the end of the list first). The first version of your code follows this algorithm:

- Traverse the list from the beginning to determine what n is.
- For $i = n, n-1, n-2, ..., 1$, traverse the list from the beginning to the i^{th} element and print it

The second version of your code looks like this, calling `printReverse` on the first node in the list.

```
class ListNode {
   int x;
   ListNode next;
   void printReverse() {
     if(next != null)
       next.printReverse();
     System.out.println(x);
   }
}
```

Why might the second version be faster?

i. temporal locality
ii. spatial locality
iii. it is faster asymptotically
iv. none of the above

Solution:
(a) i. temporal locality, (b) ii. spatial locality, (c) iii. it is faster asymptotically

Name:_____

10. (**9** points)

(a) If you take a correct parallel program and comment out the lines that call a `join` method, which ***one*** of the following is most likely to happen?

i. It could slow down because we will never have two threads execute at the same time.

ii. It could slow down because the operating system will only assign one processor to run the program.

iii. It could produce the wrong answer because the code that is supposed to combine answers from other threads will never run.

iv. It could produce the wrong answer because the code that is supposed to combine answers from other threads will run before the locations it reads hold the correct data.

(b) For ***each*** of the following, indicate whether it can be computed with:

- A parallel map operation
- A parallel reduce operation
- Neither

i. Replace each string in an array with an all-capitals version of the same string

ii. The number of strings in an array that are not the empty string

iii. The median of an array of numbers

iv. The largest index in an array of numbers that holds a negative number

(c) According to Amdahl's Law, if 25% of a program's execution time is spent on operations that cannot be parallelized, then what is the maximum speed-up parallelism can bring to the overall program execution time?

Solution:

(a) (iv)

(b) i. map
ii. reduce
iii. neither
iv. reduce

(c) 4