NAME: ________SOLUTION SET ________ NETID: _______SOLUTION SET________

CS2110 Fall 2010 Final Exam – SOLUTION SET
December 16, 2010, 2pm-3:30pm

Write your name and Cornell netid. There are 5 questions on 10 numbered pages and one extra credit problem on page 11 for a possible 6 points of extra credit (not to mention wealth beyond your dreams of avarice, as you'll see…).

- *Check now that you have all the pages.*
- *Write your answers in the boxes provided.*
- *Use the back of the pages for workspace.*
- *Ambiguous answers will be considered incorrect.*
- *Concise but correct answers will be considered correct. Answers that ramble on and make numerous false statements will lose points.*
- *The exam is closed book and closed notes.*
- *Some of the code is abbreviated to save space, or obfuscated to avoid making the problem too easy. This does not make it good coding style.*
- *If you are colorblind and don't want us to grade in a certain color ink, please make note of it at the top of every page.*
- ***Do not begin until instructed.***

You have 90 minutes. Good luck!

	1	2	3	4	5	Extra	Total
Score	/20	/16	/22	/20	/22	/6	
Grader							

1.(20 points) Sort Algorithms

(a) *(6 points)* There are three fundamental steps to the Quicksort algorithm. What are they?

1. Choose a pivot

2. Separate the elements into two sets: Greater and less than the pivot

3. Recursively Quicksort each of these sets

(b) *(5 points)* Quicksort has an average case runtime of $O(n \log n)$, but a worst case complexity of $O(n^2)$. Is this an issue in practice? Explain.

Not really. We want the pivot value to split the input vector into two equal size halves. The usual pivot is to select the value in the middle of the vector: if the input was sorted, this will split the vector; if not, it should be a pretty random value. Performance can still be poor for a hand-crafted “evil vector” but the odds of hitting such a vector of inputs “in the wild” should be pretty low.

(c) *(4 points)* Suppose you want to sort an array of (x,y) pairs in lexicographic order, i.e., sorting by the x dimension and breaking ties with the y dimension.
Instead of writing a method to compare pairs directly, you make two calls to an existing function `DIMSORT(pair_array, dimension)`, which sorts a pair array in-place by an indexed dimension:

```
DIMSORT(pairs, 1);    // sort by Y dimension
DIMSORT(pairs, 0);    // sort by X dimension
```

This can work, but what assumption does it make about the implementation of `DIMSORT`?

It assumes that DIMSORT preserves the input ordering when it sees a tie. This way the first step sorts on the tie-breaker value and the second step, which sorts on the primary value, keeps the tie-breaking sub-sort order.

(d) *(5 points)* Here is pseudo-code for **SLOWSORT**, notable for being a truly awful sorting algorithm, even in the best case:

```
function SLOWSORT(int[] A, int i, int j):
     if i >= j: return
     m = (i + j) / 2
     SLOWSORT(A, i, m)
     SLOWSORT(A, m + 1, j)
     if A[j] < A[m]:
          swap(A[j], A[m])
     SLOWSORT(A, i, j - 1)
```

d-i) (3 points) Let R(n) be the runtime of **SLOWSORT** on an array A of size n, measured as the number of times **SLOWSORT** is called. Write R(n) as a recursive expression, but don't try to simplify. *To simplify, assume n>2. Use the notation* $\lceil exp \rceil$ *to denote exp "rounded up" and* $\lfloor exp \rfloor$ *to denote exp "rounded down", where exp is an expression.*

R(n) = $1 + R(\lfloor n/2 \rfloor) + R(\lceil n/2 \rceil) + R(n-1)$

d-ii) (2 points) Normally we measure complexity for sorting in terms of the number of comparison/swap operations performed. Is this the same as R(N) from above? Explain.

It should be the same. Each invocation of SLOWSORT will do a single comparison (and perhaps, a swap) *except* in the base case, when the vector is of length 0 or 1 (caught in the first line). Since big-O complexity ignores constants, the precise number of comparison and swap operations (it can vary from 1 if we return on the first line, to 3 if we do a swap) isn't very important here, and can be ignored.

2. (16 points) True or false? Parts a,b,c, and d all refer to the same code:

```
try { stmt } catch(E1) { stuff1 } catch (E2) { stuff2 } finally {F}
```

a	[✓ T]	F	An exception of type E1 occurs while evaluating `stmt`. This causes **`stuff1; F`** to be executed.
b	T	[✓ F]	An exception of type E2 occurs while evaluating `stmt`. This causes **`stuff1; stuff2; F`** to be executed.
c	T	[✓ F]	No exception occurs. **`F`** is not executed.
d	[✓ T]	F	Suppose exception type E1 is a subtype of E2 and an exception of type E1 occurs. **`stuff1; F`** is executed

e	[✓ T]	F	C extends B, and B extends A. Object x is of type C. The value of "C instanceof A" is true.
f	[✓ T]	F	If computing something takes expected time $O(n^2)$ for an input of length n , there could be specific inputs for which the function returns in time O(1)
g	[✓ T]	F	$O(3n^4 + 2n + 7)$ is the same as $O(n^4)$
h	[✓ T]	F	$O(\log (n^3))$ is the same as O(log n)
i	[✓ T]	F	Autoboxing occurs when a primitive type such as `int` is automatically translated to a value of the corresponding reference type, such as `Integer`.
j	[✓ T]	F	A static initializer for a class is a block of code that will be executed when the class is loaded, but this won't occur until the first time the class is referenced.
k	[✓ T]	F	Given a list of objects, Java's reflection features make it possible to identify the objects that define a **`void bark(int volume)`** method and, for those that do, to invoke **`bark(10)`** for a particular instance.
l	[✓ T]	F	If Quicksort is used to sort a uniformly random vector with no repeats, the expected performance is O(n log n), but could be $O(n^2)$ if the pivot function always picks the smallest value in the vector.
m	T	[✓ F]	Same, but now "*if the pivot always picks the median value in the vector*." Note: median is a "numerical" quantity, unlike "middle" which is a location in the vector.
n	T	[✓ F]	If class C extends B, and C overrides operation x, and b is of type B, then a method call to `b.x(args)` invokes first B's version of x, then C's version of x.
o	[✓ T]	F	If you try to access the value of an unitialized Integer field in an object x of class A, a null pointer exception will be thrown.
p	T	[✓ F]	Same, but "*the value will be 0*."

3 (22 points: 12 for correctness, 5 style, 5 efficiency). Finally, a coding question! Whew!

You are working for Google-Bike on a new Android cycling app. You are given an object `m` representing a `Map`: it has nodes (objects from a class `Node`) with directed edges (from class Edge). Each node has a field `neighbors` which is of type `LinkedList<Edge>`, and each edge has a field named `target`, giving the node to which it points, and other information such as distance, slope, difficulty, etc. `Edge` and `Node` objects don't have unique identifiers, although you can add fields that you need (see below).

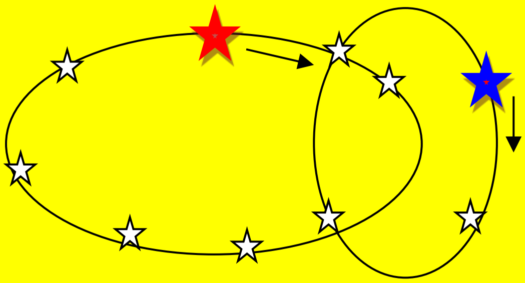
There can be multiple ways to get to the same node, via different edges (like: Over the Mountain versus Around the Mountain).

Google Bike adds a field to the nodes in a `Map` to track routes. That is, each `Node` now has an extra vector called `next[r]`. For route `r` (an `integer` id) the value is either null (if the node isn't on route `r`) or it corresponds to one of the edges in the neighbors list. We'll consider cases in which there are two routes: Route 0 and Route 1. A route starts at some start node, and you follow it node by node until you get back to the start. The Map class is also extended: it now has a vector field `start[r]` giving, for route `r` the `Node` at which that route starts.

Write a method `boolean m.compareRoutes(int routeIdA, int routeIdB)` that is called on a Map object m specifying two route ID's, and that returns true if bike routes A and B are identical except for their start points and false otherwise. Although bike routes are loops that start and end at a designated start node, routes have no other loops. Assume that the route Id numbers are within range.

If you need to add additional fields to the `Node` or `Edge` class, you must tell us precisely which class the field will be added to, what its type and name are, and how it will be used. If the default value is non-null (or non-zero, for a base type like int), you must indicate precisely how it would be initialized.

A comment about the solution: a number of students found it hard to visualize this problem until we drew pictures for them, showing two overlapping ovals. We explained (during the exam) that a "route" is like an oval made of edges: it starts and ends at some spot, and traces a kind of loop. Two routes are two ovals. The rule is that the routes are the same even if you started at different spots along the identical oval. Routes differ if the ovals differ: if even a single edge is in one, but not the other. The pictures really helped for people who found the question hard to visualize. For example, here are two different routes (we didn't show every single arrow but imagine that both are really chains of arrows).



```
boolean compareRoutes(int routeIDA, int routeIDB)
{
        // First “align” the routes by tracing route B until we
        // either loop (in which case they differ), or we encounter the
        // start of route A (in which case they are aligned)
        Node na = start[routeIDA], nb = start[routeIDB];
        do
        {
                if(nb == na) break;
                // Not at the start of route A yet, so jump to the next node
                nb = nb.next[routeIDB].target;
        }
        while(nb != start[routeIDB]);
        // Now there are two cases but we can actually treat them as a single case:
        // [case 1] If route B looped without finding the start of A, the routes differ, return false
        //              Notice that in this case, nb != na, so the “if” below catches it
        // [case 2] Otherwise, node by node, check for match and then advance both pointers
        //  If we loop all the way through route A with a match the whole way, success!
        do
        {
                // If we aren’t on the same node, return false.  Or if we are, but the
                // next edge differs for the two routes.  Here we compare the actual edge
                // objects, since there may be more than one edge from node X to node Y
                // and they aren’t the identical “route” if so.  We can use direct comparison
                // because these “pointers” would still point to the same edge object
                If(nb != na || next[routeIDA] != next[routeIDB])
                        return false;
                na = next[routeIDA];
                nb = next[routeIDB];
        }
        while(na != start[routeIDA]);
        return true;
}
```

Note: this code wasn’t the only way to solve the problem. First, the loop given above that figures out if the two rides have a shared node (the first loop) isn’t as efficient as it could be: you could walk around from startIDA the way we do but simultaneously from startIDB, and this would be faster if from startIDA to startIDB on route A is, say, 100 miles in 10,000 edges, but from startIDB to startIDA is six inches. Could happen. So you could improve the execution speed by making that loop a tiny bit more complex.

You could also do a recursive depth-first search of the map using the neighbors lists and starting at either of startIDA or startIDB. If a node is on either ride, it should be on both, and you can tell by checking (n.next[routeIDA] == n.next[routeIDB]): either both are null, or both are the same edge (this is good) or they differ (in which case the rides must differ). To avoid infinite loops you would need to add a field “nodeChecked”, initially false, setting to true when you enter your node checking logic. You get a short, elegant piece of code, but it visits every node in the whole map, hence could be very slow.

4. (20 points) Search Trees

(a) *(5 points)* Recall that an invariant is a property that always holds for a data structure. What is the invariant that distinguishes a binary search tree from an "ordinary" binary tree?

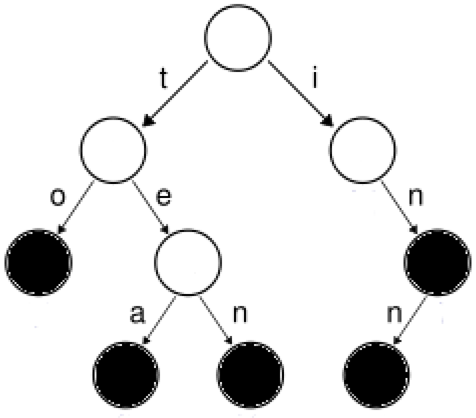
In a BST, the nodes in the left subtree all have values less than or equal to the value of the current node, and the nodes in the right subtree all have values greater than or equal to the value of the current node. Many binary trees have this property, but not all of them: to be a binary tree, one simply defines nodes that each have a value, a left child, and a right child (either of which could be null).

(b) *(5 points)* Imagine that we are implementing a delete method for a binary search tree. Deleting a value in a leaf node with no children is easy, just "delete" the node by setting the parent's reference to it to null. However, deleting an internal node this way is not recommended as it will also disconnect its entire subtree. Instead, we swap the value at the internal node to be deleted with that of a leaf node, and then delete the leaf.

Which leaf would be a good candidate to swap the internal node's value with so that the BST is still correct after the deletion?

The value associated with the right-most leaf in the left subtree, or that of the left-most leaf in the right subtree. These values preserve the BST invariant: the former is the largest value in the left subtree (hence is greater than or equal to any other value in the left subtree, and less than or equal to any value in the right subtree); the latter is the smallest value in the right subtree (hence, less than or equal to any other value in the right subtree, and greater than or equal to any value in the left subtree).

(c) *(5 points)* A **trie** is a kind of tree data structure used to store a sorted set of strings. To look up a string, begin at the root and follow the path of characters in the string. Nodes store a boolean value to indicate that the path from the root to that node represents a complete string stored in the set, and not a partial prefix. This allows a word that is a substring of another to be stored, for example, "in" and "inn" shown below. The edges in our trie will be labeled with just a single character.



Assuming a 26-character alphabet, what is the Big-O complexity of the lookup operation for a trie containing *n* strings of maximum length *m*? How does this differ from a binary search tree containing the same set of strings?

A trie of this type will have maximum depth m, hence the worst-case complexity is O(m). The average case complexity will be the same as the average string length, but this will be some constant times m, hence the big-O complexity will still be O(m).

If we build a BST with n strings, and it happens to be balanced, the lookup time will be O(log n). But if the BST isn't balanced, it could be as bad as O(n), no matter how big or small m happens to be.

(d) *(5 points)* Imagine you are implementing a spellcheck feature for a word processor. Your program needs to be able to identify misspelled words, and also propose corrections. Would a trie be a good data structure for storing a dictionary? Explain.

Yes, it could be a very compact structure. In a dictionary many words share a prefix, and a trie will represent any such words using a single path. The resulting tree will have lower depth and hence faster lookup times than a full BST for the same set of words. In effect, a trie is like a "base 26" search tree whereas a BST is a base-2 search tree (but on the other hand, the trie makes no "use" of some of the inner nodes – only the blackened in ones represent the end of a legitimate word, and there could be many non-black inner nodes. In contrast, a BST stores actual words in all the inner nodes).

Indeed, the number of nodes in a trie is potentially large than in a BST: a trie has one node per unique word (the blackened-in node) plus additional helper nodes to represent the branching points that didn't happen to also be word termination points, whereas a BST has one node per word in it. So a trie is a very squat, fat tree with potentially more nodes, but in principle, very fast lookups because of the bigger branching factor at each node (despite the fact that our *picture* shows a trie that happens to look like a binary tree).

Further improvements can be had by labeling edges in the trie with strings rather than single letters. We saw examples like this (with string labels) very briefly in our lecture on tree algorithms, but ran out of time while discussing them.

5. (22 points) The Amazing Tales of the Oaksly Service

URL shortening websites like bit.ly work much like a hashtable where a cryptic short string (e.g., the XYZ in http://bit.ly/XYZ) is the key, and a long URL is the value. You decide to start a company for yet another URL shortening service and call it oaks.ly. Using your knowledge from CS 2110, you decide to maintain the maps from short URLs to long ones entirely in an in-memory hashtable just like the ones we used in class.

(a) (6 points) Your first employee, Gary, got his CS degree in a correspondence program. He is a bit mystified as to why we need to use a hashtable. He suggests maintaining a list of (short URL, long URL) pairs instead. What are the pros and cons of a hashtable versus a list, and why is a hashtable a good choice for Oaksly?

Well, a hashtable does require a good hashing function, but will give O(1) lookup and insert times. It will perform badly if multiple keys hash to the same hashtable entry, forcing the data structure to use an overflow chaining structure to represent the values for those "collision" cases. Gary's structure is O(1) for inserts but O(n) for lookups.

(b) (6 points) Gary is amazed by the idea of a hashtable, but can't believe they really exist. Briefly explain how hashtables are implemented. (3-5 sentences)

We usually implement a hashtable as a vector of some size. The hash function is used to turn the key into a random number, and then we mod by the vector length to find the desired element. An element is set to null if nothing maps to that entry and otherwise to the value. If two or more elements map to the same vector element, we use a linked list to represent the list of values.

(c) *(5 points)* You're going to need a good hash function for your custom-built hashtable data structure. Gary learned that every character has a numerical value dictated by the ASCII encoding scheme (for example 'A' = 65), so he suggests using the sum of the ASCII values of the characters in short URL as the hash key. Is that a good idea? Explain briefly.

We'll have a lot of strings at Oaksly, yet Gary's scheme only produces a fairly small range of values, namely 52*the average short URL length. Worse, if two URLs have the same characters in different orders, Gary's function causes a collision. Basically, his idea is terrible: it "breaks" the hashtable.

(d) *(5 points)* What is the *worst possible* performance for a hashtable? What hash code function could provoke that sort of worst lookup and store operation performance for Oaksly?

The worst possible would be O(n) and the easiest way to provoke this is to use a hash function that just returns the identical value (for example, 0) for every key. We end up trying to store every value in the same vector element, and this reverts to Gary's original terrible idea from part (a).

Extra Credit. (6 points) The Oaksly Service Part II: The Empire Strikes Back

Congratulations! Gary managed to debug his hashtable-version of the Oaksly service and the company is a great success; Wired magazine already proclaimed you the new King of the Internet. Everyone on the planet uses your service. Money is pouring in faster than you can spend it. But one day, Gary shows up looking pasty and sick. The Oaksly computer will run out of memory sometime next week and when it does, your new riches will be lost. He can't add more memory: the machine is maxed out!

You rush out to Best Buy and pick up a few hundred gigabytes of file space. But how will you use files as a "backing storage" area for the Oakly data? Don't write any code; just tell us how Gary should do it. (After all, you're the boss and he just works there! This is what founding a company is all about. Maybe you'll give him a raise if he doesn't mess up.)

To make things simple, assume that a file contains objects from a special kind of class that inherits from a class called "FileObject". When you access a "file object" the disk needs to be read (if you look at the value of the object) or written (if you change the object). But otherwise these file objects are like other objects. *Hint: File I/O is slow. A good solution won't access the storage area more often than needed.*

I would treat all of main memory as a big "cache" (that is, use memoization). My disk objects would be (short-URL, long-URL) pairs, accessed via a hashing scheme, but I would have a second in-memory version of the hash table and for that version, would use the older Oaksly scheme. Keep the most commonly accessed URLs in memory, and the less common ones out on the disk.

With any luck at all, most accesses will be to just a tiny fraction of our pairs. So we rarely will need to consult the disk. Yet we do keep all the data and nothing is ever lost or forgotten.

We'll need to keep counters of access frequency (ideally, access rates over some reasonable period of time). These are easy to compute and when a new URL pair is defined, we can just keep it in memory for a while until we know its access rate. This way if a new pair gets accessed a lot it can bump some less frequently accessed pair out to the disk.

If the hash table for the full set of disk objects is too big to fit in memory, we could just use the same trick again on the hashtable vector itself: keep chunks of it in memory (namely the chunks that are being accessed most often), but have the vector as a whole be a "disk object". So we can decide precisely how much to keep in memory, and yet aren't in any sense limited by our main memory size.