CS312 Preliminary Exam II

Name:	
NetID:	(e.g., gm2d – not 112345)

Problem Number	Possible Points	Points Received	Grader
1.	15		
2.	15		
3.	12		
4.	12		
5.	12		
6.	11		
7.	12		
8.	12		
total	101		

1. [15 points, 3 each] For each of the following, tell us whether the statement is true or false, and if the statement is false, correct it so that it is true.

    a. In full SML, variables are mutable (i.e., can change value).

    b. A hash table guarantees constant time lookup.

    c. It is possible to write a functional queue implementation so that inserting an item takes constant time.

    d. We used an array in class to implement queues with O(1) insert and O(n) remove.

    e. A graph with V vertices and E edges can be represented using an adjacency matrix of size O(V).

    f. Testing proves the absence of bugs.

2. [15 points total] For each of the following, draw a box-and-pointer diagram that demonstrates what memory looks like after evaluating the given expression.

a. [4 points] `ref (ref 3, ref 4)`

b. [5 points]

```
let val x = ref 3
in
   fn (y:int) => (x := y)
end
```

c. [6 points]

```
let val x = 1
    val x = (fn (y:int ref) => y := (!y) + x)
    val x = (x, ref 4)
in
   (#1 x)(#2 x)
end
```

3. [12 points] Suppose we have an interface for directed graphs that looks like this:

```
signature GRAPH = sig
  type graph
  val new_graph     : int -> graph
  val insert_edge   : graph * int * int -> unit
  val exists_edge   : graph * int * int -> bool
  val num_out_edges : graph * int -> int
end
```

where `new_graph`(n) creates an empty graph with n vertices and no edges, `insert_edge`(g,u,v) inserts an edge into the graph between vertex u and vertex v, and `num_out_edges`(g,u) returns the number of edges from u to any other vertex in the graph. (You may assume that we never insert more than one edge between any two vertices.) Fill in the definitions for `insert_edge`, `exists_edge`, and `num_out_edges` so that the following structure implements the GRAPH interface.

```
structure Graph :> GRAPH = struct
  open Array
  type graph = (bool array) array

  fun new_graph(n:int):graph = tabulate(n, fn i => array(n,false))

  fun insert_edge(g:graph,u:int,v:int) =




  fun exists_edge(g:graph,u:int,v:int) =




  fun num_out_edges(g:graph,u:int) =









end
```

4. [12 points] What is the running time (in terms of the number of vertices n in the graph) for each of the operations you implemented?

a. `insert_edge:`

b. `exists_edge:`

c. `num_out_edges:`

5. [12 points] What would the running times be if we used adjacency lists (with no duplication of edges) to represent the graph?

a. `insert_edge:`

b. `exists_edge:`

c. `num_out_edges:`

6. [11 points] Describe how you could implement the graph interface so that inserting an edge, testing for the existence of an edge, or calculating the number of edges leaving a vertex would all be constant-time operations. Your answer should include an SML type definition for your data structure, an explanation as to how it would be used to represent the graph, and an explanation as to why each operation would be constant time.

7. [14 points] Consider the following psuedo-code:

```
fun array_crunch(a: int array):unit =
  if (Array.length a) > 1 then
     let val m:int = process(a)
     let val (b:int array,c:int array) = split(a,m)
     in
        array_crunch(b);
        array_crunch(c);
        merge(b,c,a);
     end
  else ()
```

You may assume that process(a) runs in time O(lg(length a)), that split(a,m) runs in time O(length(a)) and produces two arrays that are half as big as a, and that merge(b,c,a) runs in time O(n).

a. Write a set of recurrence equations that describe the time used by this program in terms of the length of the input array.

b. Solve the recurrence, using big-O notation for your solution.

8. [12 points] Suppose we are given a hash table and we wish to insert the keys 3, 13, 31, 21, and 15 when the hash function is (fn x => x mod 5).

a. Draw a picture of the hash table that results if we use linear probing to resolve conflicts.

b. Draw a picture of the hash table that results if we use double hashing to resolve conflicts with a second has function (fn x => (x+1) mod 5).