UNIVERSITY OF TORONTO

FACULTY OF APPLIED SCIENCE AND ENGINEERING

FINAL EXAMINATION, APRIL 2011

CSC 190 H1S — ALGORITHMS, DATA STRUCTURES, AND LANGUAGES

Calculator Type: ***None***

Examination Type: X

Examiner(s): François Pitt and Kante Easley

Student Number:	⎿⏊⏊⏊⏊⏊⏊⏊⏊⏌
Family Name(s):	________________________
Given Name(s):	________________________
Lecture Section:	☐ LEC 01 (with F. Pitt) ☐ LEC 02 (with K. Easley)

Do ***not*** *turn this page until you have received the signal to start.*
In the meantime, please read the instructions below *carefully*.

This final examination paper consists of 5 questions on 18 pages (including this one), printed on both sides of the paper. *When you receive the signal to start, please make sure that your copy is complete, fill in the identification section above, and write your student number where indicated at the bottom of every odd-numbered page (except page 1).*

Answer each question directly on this paper, in the space provided. If you need more space for one of your solutions, use one of the extra "blank" pages at the end of the examination and *indicate clearly the part of your work that should be marked.*

Write up your solutions carefully! In particular, use notation and terminology correctly and explain what you are trying to do—part marks *will* be given for showing that you know the general structure of an answer, even if your solution is incomplete.

When writing code, comments are **not** required *except where indicated*, although they may help us mark your answers. They may also be worth part marks if you can't figure out how to write the code.

If you are unable to answer a question (or part), you will get 20% of the marks for that question (or part) if you write "I don't know" and nothing else—you will *not* get those marks if your answer is completely blank, or if it contains contradictory statements (such as "I don't know" followed or preceded by parts of a solution that have not been crossed off).

MARKING GUIDE

# 1:	______/22
# 2:	______/14
# 3:	______/12
# 4:	______/12
# 5:	______/20
TOTAL:	______/80

Good Luck!

Question 1. [22 MARKS]

Part (a) [2 MARKS]

Complete the line of C code below to allocate enough memory for 10 `ints`, and store the address of this block of memory in variable `array` (assume all necessary `#include` directives).

```
int *array =
```

Part (b) [3 MARKS]

Complete function `center` below, according to its comment.

```
struct point { double x, y; };
struct rectangle { struct point upper_left, lower_right; };

/*  Return the center of rectangle r.  */
struct point center(struct rectangle r)
{











}
```

Part (c) [3 MARKS]

Complete function `destroy` below, according to its comment.

```
struct node { int data; struct node *next; };

/*  Free the memory for every node in the linked list starting at node 'first'.  */
void destroy(struct node *first)
{











}
```

Question 1. (CONTINUED)

Part (d) [2 MARKS]

What is the output of the following code?

```
#define SQUARE(x) ((x)*(x))

int i = 0;
while (i < 5)
    printf("%d\n", SQUARE(i++));
```

Part (e) [2 MARKS]

Show the output of the following code.

```
void perchance_swap(void **a, void **b)
{
    void **temp_pptr = a;
    **(float**)a = **(float**)b;
    b = temp_pptr;
}

...
float *a = malloc(sizeof(float));
float *b = malloc(sizeof(float));
*a = 1;
*b = 2;
perchance_swap((void**)&a, (void**)&b);
printf("%g %g\n", *a, *b);
...
```

Part (f) [2 MARKS]

Find and fix the bug (there is exactly one) in the following code.

```
typedef struct string_buf {
    unsigned capacity;
    char *string;
} *string_buf_t;

...
string_buf_t buf;
buf->capacity = 100;
buf->string = malloc(buf->capacity);
strcpy(buf->string, "Hi there!");
...
```

Question 1. (CONTINUED)

Part (g) [2 MARKS]

- **True** or **False**: Building a heap out of n elements takes time $\mathcal{O}(n \log n)$ in the worst-case? ______
- **True** or **False**: Building a heap out of n elements takes time $\Omega(n \log n)$ in the worst-case? ______

Part (h) [2 MARKS]

In order to maintain $O(1)$ *average* run time for accessing an element in an open-addressed hash table, what are two properties that your hash table and hash function must have?

Part (i) [4 MARKS]

Add code below so that the call to `qsort` will sort the array of students (`classlist`) in increasing order of their student number.

```
struct student {
    long number;
    char *name;
};
```

```
struct student classlist[200] = { /*...list of student names and numbers...*/ };

qsort(classlist, 200, sizeof(struct student), compare);
```

Question 2. [14 MARKS]

Consider the following Stack ADT (adapted from class), defined in a file named "stack.h":

```
#ifndef STACK_H
#define STACK_H

#include <stdbool.h> /* for bool */

/*  The type used to represent a stack -- defined in the implementation file.
 */
typedef struct stack *stack_ptr_t;

/*  Create and return a new Stack.
 *  Parameters   :  none
 *  Return value:  a new Stack (NULL in case of error)
 *  Side-effects:  memory has been allocated for a new Stack
 */
stack_ptr_t stack_create(void);

/*  Free all memory allocated for a Stack.
 *  Parameters   :  s != NULL: a Stack
 *  Return value:  none
 *  Side-effects:  all memory allocated for s has been freed
 */
void stack_destroy(stack_ptr_t s);

/*  Return whether or not a Stack is empty.
 *  Parameters   :  s != NULL: a Stack
 *  Return value:  true if s is empty; false otherwise
 *  Side-effects:  none
 */
bool stack_is_empty(const stack_ptr_t s);

/*  Add an item to the top of a Stack.
 *  Parameters   :  s != NULL: a Stack;  x: the item to add to s
 *  Return value:  none
 *  Side-effects:  terminates the program in case memory allocation fails
 */
void stack_push(stack_ptr_t s, void *x);

/*  Remove and return the item on top of a Stack.
 *  Parameters   :  s != NULL: a Stack
 *  Return value:  the item stored on top of s
 *  Side-effects:  terminates the program in case the stack is empty
 */
void *stack_pop(stack_ptr_t s);

#endif/*STACK_H*/
```

Question 2. (CONTINUED)

Assuming that your code will be compiled along with a file **stack.c** that implements **stack.h**, complete the code below (and on the next page) so that function **pre_order_print** carries out the same work as function **pre_order_print_rec**, but *non-recursively.*

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "stack.h"

/*  A struct for "ternary" trees (where each node has at most 3 children).  */
typedef struct tree_node {
    long elem;    struct tree_node *left, *middle, *right;
} tree_node_t;

/*  An item on the stack used to simulate the recursive function -- WRITE THIS.  */
typedef struct stack_item {



} stack_item_t;

/*  Allocate memory for a new stack_item and return a pointer to it -- WRITE THIS --
    include appropriate parameters.  */
static stack_item_t *new_item(                                              )
{







}

/*  Free the memory allocated for a stack_item -- WRITE THIS.  */
static void free_item(stack_item_t *item)
{



}

/*  Helpers to make the stack functions easier to use.  */
static void push(stack_ptr_t s, stack_item_t *x) { stack_push(s, x); }
static stack_item_t *pop(stack_ptr_t s) { return (stack_item_t *) stack_pop(s); }
```

CONT'D...

Question 2. (CONTINUED)

```
/*  The recursive function.  */
void preorder_print_rec(tree_node_t *root)
{
    if (root == NULL)  return;
    printf("%ld\n", root->elem);
    preorder_print_rec(root->left);
    preorder_print_rec(root->middle);
    preorder_print_rec(root->right);
}

/*  Write this function to do the same thing as preorder_print_rec, non-recursively.  */
void preorder_print(tree_node_t *root)
{
```

```
}

/*  Code would go here (in a main function) to create different ternary trees and call
 *  preorder_print_rec and preorder_print on each one, to check that both functions do
 *  the same thing.
 */
```

Question 3. [12 MARKS]

Consider a binary **min** heap with the following contents:

▨	1	3	2	8	4	7	10	9	12	5					

Part (a) [2 MARKS]

Draw the tree structure for the heap above.

Part (b) [4 MARKS]

Complete the function below according to its specification (assume all necessary `#include` directives).

```
/*  Return true if and only if the supplied array is a min heap.
 *  Parameters and preconditions:
 *      array != NULL: a pointer to an array of comparable elements, each of type void *
 *      size >= 0: the number of elements in the array
 *      cmp != NULL: a pointer to a comparison function for array elements, that returns
 *          < 0, == 0, or > 0 as to the first element is less than, equal to, or greater
 *          than the second element
 *  Return value:  true if the array is a min-heap; false otherwise
 *  Side-effects:  none
 */
bool is_min_heap(void **array, size_t size, int (*cmp)(void*, void*))
{




}
```

Question 3. (CONTINUED)

Part (c) [3 MARKS]

Show the contents of the heap from the previous page, after inserting the values 0 and 6, in that order. Draw the heap tree after each insertion, and fill in the final content of the heap array below.

▨															

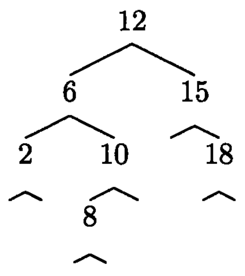
Part (d) [3 MARKS]

Show the contents of the heap after removing *two* values **from the original heap** (at the top of the previous page). Draw the heap tree after each removal, and fill in the final content of the heap array below.

▨															

Question 4. [12 MARKS]

Consider the following binary tree:



Part (a) [2 MARKS]

In what order are the nodes printed for an in-order and post-order print of the tree above?

- `in-order:`
- `post-order:`

Part (b) [4 MARKS]

Complete the function below according to its specification.

```
typedef struct node {
    void *data;
    int height;
    struct node *left, *right;
} tree_node_t;

/*  Helper function.  */
int max(int x, int y) { return x > y ? x : y; }

/*  Compute and return the height of the tree rooted at root.  At the same time, set the
 *  height field of each node equal to the height of the subtree rooted at that node --
 *  use -1 for the height of an empty tree (with root == NULL).
 */
int set_height(tree_node_t *root)
{









}
```

Question 4. (CONTINUED)

Part (c) [2 MARKS]

Treating the tree on the previous page as a BST, insert a node with value 9 and redraw the tree below, showing where the new node is inserted.

Part (d) [4 MARKS]

Now, treating the *original* tree (without any new nodes added) as an AVL tree, insert a node with value 9 and redraw the tree below. Clearly show where the new node is inserted, and show all rotations required to re-balance the tree.

Question 5. [20 MARKS]

A "multiset" is like a set, except that elements are allowed to occur multiple times. For example, $\{-2, 7.1, .4, 7.1\}$ is **different** from $\{7.1, .4, -2\}$: 7.1 appears twice in the first multiset, but only once in the second one. Consider the following file "`multiset.h`".

```
/*  A basic multiset ADT.
 */
#ifndef MULTISET_H
#define MULTISET_H

#include <stddef.h> /* for size_t */

/*  The type of a multiset -- to be defined in the implementation.
 */
typedef struct multiset *multiset_ptr_t;

/*  Create and return a new multiset.
 *  Return value:  a new multiset (NULL in case of error)
 *  Side-effects:  memory has been allocated for a new multiset
 */
multiset_ptr_t multiset_create(void);

/*  Free all memory allocated for a multiset.
 *  Parameters   :  set != NULL: a multiset
 *  Side-effects:  all memory allocated for set has been freed
 */
void multiset_destroy(multiset_ptr_t set);

/*  Return the number of occurrences of an element in a multiset.
 *  Parameters   :  set != NULL: a multiset;  elem: an element
 *  Return value:  the number of times elem appears in set (possibly 0)
 */
size_t multiset_contains(const multiset_ptr_t set, const void *elem);

/*  Insert an element into a multiset.
 *  Parameters   :  set != NULL: a multiset;  elem: an element
 *  Side-effects:  set contains one more occurrence of elem than before
 */
void multiset_insert(multiset_ptr_t set, const void *elem);

/*  Remove an element from a multiset.
 *  Parameters   :  set != NULL: a multiset;  elem: an element
 *  Side-effects:  set contains one less occurrence of elem than before
 *                 (but no less than 0 occurrences)
 */
void multiset_remove(multiset_ptr_t set, const void *elem);

#endif/*MULTISET_H*/
```

Question 5. (CONTINUED)

Part (a) [10 MARKS]

Describe two different data structures that could be used to implement the multiset ADT (you will implement one of them in the next part of this question). For each data structure,

- describe how the multiset's elements are stored in memory (*i.e.*, how you organize the data)—*your code must be able to handle any number of elements* (limited by the computer's memory, of course),
- state the worst-case running time of each function—*it's okay if your implementation is inefficient*,
- describe one main *advantage* of your data structure,
- describe one main *disadvantage* of your data structure,
- **do NOT write any code for this part!**

Question 5. (CONTINUED)

Part (b) [10 MARKS]

On this page and the next, write complete code for a file "`multiset.c`" that implements the multiset ADT, using one of your data structures from the previous part. *Hints:*

- Write your code "top-down": start with a high-level outline (as comments) for each function, then fill in code to carry out each step of your outline (creating and calling helper functions as appropriate).
- *Do not write a* `main` *function or any testing code!* Just implement the functions from `multiset.h`, along with any required types and helper functions.
- Your code will be marked on its design as well as its correctness (but not its efficiency).
- Don't forget appropriate `#include` statements, constants, and types.

Question 5. (CONTINUED)
Part (b) (CONTINUED)

[Use the space on this page for scratch work, or for any part of an answer that did not fit elsewhere. ***Clearly label each answer with the appropriate question and part number.****]*

[Use the space on this page for scratch work, or for any part of an answer that did not fit elsewhere. ***Clearly label each answer with the appropriate question and part number.****]*