Final Practice Set

CS 136 Spring 2016

PREPARED BY CS 136 ISAS

Disclaimer: This set of questions is intended for practice only! It is NOT meant to be a representation of the actual final exam

University of Waterloo

Instructions: The instructions and the cover for the final exam will be posted closer to the examination date

1. (8 points) Multiple Choice: circle the most correct answer.

(a) (2 points) What strategy can we use to improve the average run-time when dynamically allocating an array of unknown length?

a. Wrapper Strategy
b. Functional Strategy
c. Augmentation Strategy
d. Doubling Strategy

(b) (2 points) What is the difference between a C string and an array of **`char`**s in C?

a. C strings are always stored in read-only memory, whereas an array of **`char`**s do not have this restriction.
b. An array of **`char`**s cannot be initialized using the double quote syntax (i.e. `char a[] = "Hello, World"`).
c. An array of **`char`**s stores its length.
d. C strings are null terminated.
e. There is no difference.

(c) (2 points) Which approach permits us to write an $O(1)$ `add_to_back` function for a singly linked list?

a. Use the augmentation strategy.
b. Use recursion in our `add_to_back` function.
c. Use iteration in our `add_to_back` function.
d. Use the wrapper strategy.

(d) (2 points) What approach would be able to determine the height of the BST in $O(1)$ time?

a. Augment the BST node to store the height of the subtree and increase it during each insertion.
b. Recurse through the longest subtree.
c. Count the number of items stored in the tree, returning $\log_2(n) + 1$ where n is the number of items stored.
d. It is not possible to implement an O(1) height function for a BST.

2. (6 points) Short Answer: Provide a scenario for the following ADTs. For each question below, provide a scenario where the specified ADT may be utilized. Briefly justify your answer.

(a) (2 points) Dictionary

(b) (2 points) Stack

(c) (2 points) Queue

3. (6 points) Multiple Choice: circle the answer with the lowest correct upperbound. Consider the runtime in the worst case.

(a) (2 points) What is the runtime of the following function in terms of n, where n is the length of `lon`?

```
(define (list-max lon)
  (cond
    [(empty? lon) -inf.0]
    [(> (first lon) (list-max (rest lon))) (first lon)]
    [else (list-max (rest lon))]))
```

a. $O(n)$
b. $O(n^2)$
c. $O(2^n)$
d. $O(n \log n)$

(b) (2 points) What is the runtime of the following function in terms of n, where n is the string length of `str`?

```
char lastletter(char *str) {
  char retval = '\0';

  for (int i = 0; i < strlen(str); i++) {
    if (((str[i] >= 'a') && (str[i] <= 'z')) ||
        ((str[i] >= 'A') && (str[i] <= 'Z'))) {
      retval = str[i];
    }
  }

  return retval;
}
```

a. $O(1)$
b. $O(\log n)$
c. $O(n)$
d. $O(n^2)$

(c) (2 points) What is the runtime of the following function in terms of n, where n is the length of `lon`?

```
(define (part-sum lon)
   (cond
      [(empty? lon) 0]
      [(empty? (rest lon)) (first lon)]
      [(empty? (rest (rest lon))) (+ (first lon) (second lon))]
      [else (+ (first lon) (second lon) (third lon))]])
```

a. $O(1)$
b. $O(\log n)$
c. $O(n)$
d. $O(n^2)$

4. (35 points) Answer the short questions below in one or two sentences.

(a) (5 points) Recall that the five regions in memory are: code, read-only, global data, stack, and heap. Read the following code:

```
#include <string.h>

char *myvar;

int main(void) {
    char *str = "CS 136";
    char **c = &str;

    myvar = str;
    char *newstr = strdup(str);
}
```

At the end of this program, which region of memory is each of the pointer referring to?

- str:
- c:
- myvar:
- newstr:
- main:

(b) (3 points) Racket has functions that are equivalent to `free` and `malloc` functions found in C. However, they are rarely used in Racket. Why is the use of `free` and `malloc` less prevalent in Racket code?

(c) (3 points) Why is it invalid for a function to return a pointer to a variable in its own stack frame?

(d) (3 points) We saw in lecture how we could use the wrapper strategy to obtain an $O(1)$ length function for linked lists. Starting with the linked list structure below, how could you write an $O(1)$ length function using the augmentation strategy (where each node is augmented)? Note: You do not need to write any code, just explain how you could do this.

```
struct llnode {
        int item;
        struct llnode *next;
};
```

(e) (3 points) Why should we use always give `malloc` a `sizeof(type)` argument, rather than an integer? Example: `malloc(sizeof(int))` as opposed to `malloc(4)`

(f) (3 points) Sort the following list of orders in Big O notation, from smallest to largest:
$O(2^n), O(n), O(\log(n)), O(n^2), O(1), O(n\log(n))$

(g) (3 points) Consider the following C code:

```
int i = 2;
int *q = malloc(sizeof(int));
q = &i;
```

Have we leaked memory yet? If so, explain where and if not, write the necessary C code to free all of the dynamically allocated memory.

(h) (3 points) What's the difference between a data structure and an abstract data type?

(i) (3 points) What's the difference between a function declaration and a function definition? Provide an example of both.

(j) (3 points) Describe what a pointer is in C. You may find it useful to use a picture of the memory model. Give an example of where pointers are useful.

(k) (3 points) What's an opaque structure in C and give an example (in code)? Provide an example of how opaque structures impact modularization.

5. (6 points) In the following code sample, identify (circle) three errors, and briefly explain why they are erroneous.

```
struct myarray {
    void **array;
  int size;
}

typedef struct myarray *MyArray;

MyArray construct_MyArray(int size) {
  MyArray newArray = malloc(sizeof(MyArray))
  newArray->size = size;
  newArray->array = malloc(sizeof(void *) * size);

  return newArray;
}

void destroy_MyArray(MyArray myArray) {
  if (myArray) {
    free(myArray);
  }
}

void insert_MyArray(MyArray myArray, int index, void *data){
  assert(myArray);
  if (myArray->size >= index){
    myArray->array = realloc(sizeof(void *) * (index + 1));
    myArray->size = index + 1;
  }
  else {
    myArray->size++;
  }
  myArray->array[index] = *data;
}
```

6. (10 points) Consider the following function declaration:

```
#include <stdbool.h>

struct bstnode {
  int key;
  int val;
  struct bstnode *left;
  struct bstnode *right;
};

// bst_search(bst, i) determines if i exists in bst.
// requires:  bst is not NULL and bst is balanced
// time: O(log(n)), where n is the number of nodes in bst.
bool bst_search(struct bstnode *bst, int i);
```

(a) (5 points) Write the implementation for `bst_search` recursively.

(b) (5 points) Write the implementation for `bst_search` iteratively.

7. (10 points) Write the C code for the following function:

```
// filter(arr, n, pred) removes all elements from arr for which pred
//   evaluates to false and returns the size of the filtered array.
// requires: arr is a dynamically allocated array of size n
// effects: arr is mutated and resized to the minimum size to hold the
//   filtered elements.
// time: O(n)
int filter(int **arr, int n, bool (*pred)(int));
```

8. (10 points) Write the implementation for the following header file.

```
#include <stdbool.h>

// reverse_string(str) reverses the order of all the characters
//   inside of str.
// requires: str is valid
// effects: The order of str is mutated
void reverse_string(char *str);

// censor(str, word) censors all occurrences of word
//   in str.
// requires: str and word are not NULL, and str and word
//   are valid C-strings
// effects: allocates a new block of memory on the heap that
//   is the same size as the block of memory that str
//   points to. Replaces all occurrences of word in str
//   with asterisks ('*'). If there are nested occurrences,
//   censor the first one (i.e. word is "bob" and str is
//   "bobob" the result is "***ob").
//   caller must free resulting string.
// time: O(nm), where n is the string length of str and m is
//   the string length of word.
char *censor(char *str, char *word);
```

9. (15 points) In this question you will implement an ADT for controlling a robot. Initially, the robot is at the point (0,0). The robot can be given commands to move with store_instruction(), but will only execute them when go() is called. When go() is called, the robot will execute each action in the same order in which they were given via store_instruction(). After executing each action, go() should print out the current position followed by a newline ('\n') character (i.e., "(3,4)\n").

The instructions that can be sent to the robot are 'u', 'd', 'l', and 'r', which move the robot up, down, left, and right, respectively. Write the implementation for the header file given below.

```
struct robot;

typedef struct robot *Robot;

// create_Robot() creates a new robot.
// effects: allocates a new block of memory for a new robot. The
//              new robot is at the origin facing north.
//              caller must make a call to destroy_Robot.
// time: O(1)
Robot create_Robot(void);

// destroy_Robot(r) destroys the robot.
// requires: r is a valid robot
// effects: frees all of the memory associated with the robot, r.
// time: O(n), where n is the number of stored instructions.
void destroy_Robot(Robot r);

// store_instruction(r, i) stores the instruction for the robot.
// requires: r is a valid robot and i is one of { 'u', 'd', 'l', 'r' }
// effects: allocates a new block of memory for an instruction.
// time: O(1)
void store_instruction(Robot r, char i);

// go(r) executes all of the stored instructions for the robot.
// requires:  r is a valid robot
// effects:   executes all of the stored instructions
//            frees all instructions.
// time: O(n), where n is the number of queued instructions.
void go(Robot r);
```