Some CPSC 259 Sample Midterm and Final Exam Questions (Part 2)

DON'T LOOK AT THESE SOLUTIONS UNTIL YOU'VE MADE AN HONEST ATTEMPT AT ANSWERING THE QUESTIONS YOURSELF.

1. {3 marks} Find the errors in the following code fragment. List each erroneous statement and briefly describe the error in it.

```
#define NUM  777

int main(void)
{
  int x = 10;
  int *p = NUM, *q = NULL;

  *p = x;
  x  = *q;
  &x = NUM;

  ...
}
```

ANSWER:

a) Assigning 777 to pointer `p` is almost certainly wrong, since 777 isn't likely to be a valid address.
   - Closely related to (a): `*p` – dereferencing `p` is likely to crash the program

b) `*q` – dereferencing `q` is likely to crash the program

c) You cannot assign a value to the address of `x` because `x`'s address is fixed at compile time.

2. {4 marks} Find the errors in the following function and give a brief explanation for each error.

```
/* Allocate an integer array of size MAXLEN, and initialize all
elements to zero.  Then, return a pointer (i.e., address) to
element [n] in the array, where 0 <= n <= MAXLEN (else return
NULL). */

int * makeArray(int n)
{
   int  a[MAXLEN];
   int  i;

   for (i = 0; i <= MAXLEN; i++)
        a[i] = i;
```

```
    if (n >= 0  &&  n <= MAXLEN)
        return  a + n;

    return NULL;
}
```

ANSWER:

a) The first "<=" should be "<"; otherwise, we will go one spot past the end of the array. This could crash the program, or cause other "unpredictable" errors (when other memory/variables are overwritten).

b) same idea with the second "<="

c) Upon leaving the function, array "a" and all its contents are destroyed (no longer available) because they were allocated locally for the life of the function only. Thus, they will not be assessable from the main program once the `makeArray` function ends. This is a dangling pointer problem.

d) Even if the memory were allocated permanently (via `malloc` in part (c)), the returned offset into the array will be out of bounds if `n == MAXLEN`. Thus, the "`return  a + n;`" statement is incorrect.

3. {5 marks} Write a set of statements that creates a dynamic array of 100 integers, and sets all its element values to 100.

ANSWER:

```
int * intPtr;
int   k;

intPtr = (int *) malloc( 100 * sizeof(int) );
if (intPtr == NULL)
{
     /* error processing, terminate */
}
for (k=0; k < 100; k++)
     intPtr[k] = 100;
 /* *(intPtr + k) = 100; */    /* another valid way */
```

4. {5 marks} Suppose the nodes of a linked list structure are defined as follows:

```
struct node
{
  int           value;
```

```
  struct node *  next;
};
```

Define a function `length` which takes a pointer to the start of the linked list (of nodes) and returns the number of items that are in the list.

For instance, if `list` is the list (3, 9, 5, 6) then `length(list)` returns 4.

ANSWER:

```
int length(struct node * head)
{
      int count = 0;

      while (head != NULL)
      {
            count++;
            head = head->next;
      }

      return count;
}
```

5. {8 marks} Suppose the nodes of a doubly linked list structure are defined as follows:

```
struct node
{
   int           item;
   struct node * next;
   struct node * prev;
};
```

Write a function `concat` which concatenates two given lists (the first node of the second list will follow the last node of the first list) and returns the new list. Note that `concat` does not create new nodes; it just rearranges the links of some existing notes. Assume that the pointers all refer to the head of their respective list.

```
struct node *  concat( struct node * list1, struct node * list2 )
{
```

ANSWER:

```
/* We'll always return the head of the combined list.
   Note that it was changed if list1 was empty and list2
   wasn't.  The caller will have to manage the tails. */
```

```
struct node *  concat( struct node * list1, struct node * list2 )
{
      struct node * save_head_list1, * last_node;

      if (list1 == NULL  &&  list2 == NULL)
            return NULL;

      if (list1 == NULL)
            return list2;

      if (list2 == NULL)
            return list1;

      save_head_list1 = list1;
      while (list1 != NULL)
      {
            last_node = list1;
            list1 = list1->next;
      }

      last_node->next = list2;
      list2->prev = last_node;

      return  save_head_list1;
}
```

6. {3 marks} Suppose that we use a linked list to represent a queue and that in addition to the `enqueue` and `dequeue` functions (i.e., functions to add and remove elements from the linked list), you want to add a new operation to the queue that deletes the last element of the queue. Which linked structure do we need to use to guarantee that this operation is also executed in constant time? Justify your answer.

ANSWER:

We need to use a doubly-linked list, so that we can `dequeue` nodes from either end of the list in O(1) time.

7. {7 marks} Suppose you have a stack ADT (i.e., an Abstract Data Type that includes operations to maintain a stack).

a) Describe in words (no code) how you could implement a queue's `enqueue` and `dequeue` operations using two stacks. Also, provide the Big-O complexity figures.

ANSWER:

Use one stack as the one to hold all the entries in LIFO order, as usual. Let's call this Stack 1, and we'll maintain a pointer to its tail (only), as usual.

To dequeue from Stack 1: If we want to remove the node from the front of the list, we'll need to pop off all the nodes from Stack 1 and push them onto Stack 2. The final node to be popped off of Stack 1 is the one we want. This whole operation takes O(*n*) time, due to 2 passes through the list: once to dequeue the nodes from Stack 1, and once to enqueue the nodes onto Stack 2.

Finally, we can return all the nodes to Stack 1, which is again an O(*n*) operation for the reasons just described. Thus, we will use Stack 1 as the "master" list, and just use Stack 2 as the work list.

To enqueue to Stack 1: Just push the new node onto Stack 1. This is an O(1) operation.

b) Using this implementation, describe a linear time algorithm for reversing a queue.

ANSWER:

To reverse a queue: rename Stack 1 to Stack 2. Then, pop off all the nodes from Stack 2 and push them all onto Stack 1. For example:

Stack 1: 1, 2, 3
Stack 2:

becomes ...
Stack 2: 1, 2, 3
Stack 1:

becomes ...
Stack 1: 3, 2, 1
Stack 2

This is done in O(*n*) steps.