Quiz 1 Review: Answers to Practice Problems

Computer Science 50
December 9, 2008

1. `bool isOdd(int n);`

2. ```
   bool isOdd(int n) {
       return (n % 2) ? true : false;
   }
   ```

3. ```
   bool isOdd(int *n) {
       return (*n % 2) ? true : false;
   }
   ```

4. The result is: 16, and the value of x is 10 here

5. ```
   first = 0;
   last = size - 1;

   while(first < last)
        middle = avg(first, last)
        if the target is less than array[middle]
             last = middle - 1;
        else if the target is greater than array[middle]
             first = middle + 1;
        else
             return true // Why do we return true here?
   if array[first] is the target // Why do we make this check?
        return true
   return false
   ```

6. Three. The first eliminates 4, 6, 9, 11, 14, 18. The second eliminates 21, 25, 29. On the third pass, first = 9, last = 10, middle = 9. As array[9] == 32, the algorithm returns TRUE.

7. It prints out the string, character by character, beginning at foo and ending when the NULL character is reached.

8. O(n)

9. No. This algorithm prints out the string backwards.

10. They are the same because in both cases, the algorithm must traverse all of the unsorted array. In uni-directional bubble sort the endpoint only moves from one direction; in bi-directional bubble sort the endpoints move from both directions. This gives the illusion of bi-directional bubble sort being faster, but since bi-directional bubble sort makes n passes in both directions in the worst case, and $O(n)$ comparisons per pass, therefore they both actually run in O(n^2).
   ```

11.
```
double average() {
  double currVal = 0, sumTotal = 0;
  int numValues = 0;

  if((currVal = GetDouble()) == SENTINEL)
      return 0;

  do {
      sumTotal += currVal;
      numValues++;
      currVal = GetDouble();
  } while(currVal != SENTINEL);

  return (sumTotal / numValues);
}
```

The algorithm runs in O(n), because adding one more value to the list for the average only increases the amount of work needed to be done by one unit of time.

12. (a) 5 2 1 6 7 3 4 8

    (b) 2 1 5 6 3 4 7 8

    (c) 1 2 5 3 4 6 7 8

13.
```
unsigned long fib(unsigned long n) { // Does not error-check case where n < 0
    if (n <= 1) {
        return n;
    } else {
        return fib(n-1)+fib(n-2);
    }
}
```

14. The program will terminate, because the stack will fill up with copies of `another_level()`, and eventually you will run out of memory.

15. `ptr = &i;`

16. 2

17. `double *doub_array = malloc(40 * sizeof(double));`

18.
- `#define` statements do not end in a semicolon
- Semicolon missing from array declaration
- The first `for` loop will run off the end of the array
- The second `for` loop makes use of an undeclared variable
- The second `for` loop uses `i` where `j` is likely intended
- Others may be possible. Check with your TF!

19.
```
struct class_t{
    int num_students;
    char *instructor;
    float cue;
};
```

```
struct class_t CS50;
```

20. `CS50.instructor`

21. `CS50->num_students` or `(*CS50).num_students`

22. A quick format will only delete the tables that describe where the data is, as an effort to save time. A complete format, however, will write over all of the data with random 1s and 0s, so the old data will be sufficiently removed, and you will not need to worry about the eBay buyer finding old credit card numbers (which they might have been able to do by scanning the disk, much in the same way you scanned the .raw file in PS5).

23. `FILE *infile = fopen("input.txt", "r");`

24. `FILE *outfile = fopen("output.txt", "w");`

25. `unsigned char`

26. `typedef enum {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC} months;`

27. 68

28. 7 coins: 1x100c, 1x25c, 1x10c, 1x5c, 3x1c

29. 6 coins: 1x100c, 2x20c, 3x1c

30. `"r"` opens the file in read mode, while `"w"` opens the file in write mode. This means that sourceFile can only be read from, and that destinationFile can only be written to.

31. `fopen(...)` returns a pointer to a file. If opening the file fails, it returns `NULL`.

32. In line 21, up to 1000 bytes of data is read from `sourceFile` into `buffer`. `n` is the number of bytes read into buffer. In line 22, n bytes of data is written from `buffer` into the file pointed to by `destinationFile`.

33. The first time `destinationFile` was opened was in write mode. Now we want to read from it, so we have to close it, and reopen it in read mode.

34. ```
    void DeleteList(node * head) {
         node * current = head;
         node * next;
         while (current != NULL) {
              next = current->next;
              free(current);
              current = next;
         }
    }
    ```

35. There are a few ways to implement this function. One strategy is that you could make a new list and transfer nodes one by one. Another strategy is to iterate through the list rearranging pointers. The first is a bit trickier, so its implementation is provided below. Again, it assumes that it is passed a pointer to the first element of the list
    ```

```
node * ReverseList(node * head) {
     node * result = NULL;
     node * current = head;
     node * next;
     while (current != NULL) {
          next = current->next;
          current->next = result;
          result = current;
          current = next;
     }
     return result;
}
```

36. The error is in the segment that checks for insertion in the middle. The lines `predptr->next = newptr;` and `newptr->next = predptr->next;` need to be switched. Otherwise we are setting `newptr->next` to `newptr`!

37. This code will not compile because we have used a `.` in every place we should have used a `->`. Remember that we use `->` with pointers to structures and `.` with actual structures.

38. The following implementation of `void delete(node * ele)` takes as its argument a pointer to the node which should be removed.

```
void delete(node * ele)
{
   if (ele->prev == NULL)
       head = ele->next;
   else
       ele->prev->next = ele->next;

   if (ele->next == NULL)
       last = ele->prev;
   else
       ele->next->prev = ele->prev;

   free{ele);
}
```

39. In the following implementation of `int size(node * treetop)` the only argument is a pointer to the top of the tree.

```
int size(node * treetop) {
  if (treetop==NULL) {
    return(0);
  } else {
    return(size(treetop->left) + 1 + size(treetop->right));
  }
}
```

40. The lines

```
printTree(node->left);
printTree(node->right);
printf("%d ", node->data);
```

should read

```
printTree(node->left);
printf("%d ", node->data);
printTree(node->right);
```

This is because everything to the left of the current node is less than it, and everything to the right is more.

41. Solution omitted...a rote algorithm will generate this solution relatively quickly, but would be overly cumbersome to print here.

42. The following function takes as its argument a pointer to the head of the tree.

```
void printAsArray(node * node) {
  if (node == NULL) return;
  printf("%d ", node->data);
  printAsArray(node->left);
  printAsArray(node->right);

}
```

43. The probability of no collision is

$$(m-1)/m * (m-2)/m * ... * (m-n+1)/m$$

so the probability of a collision is

$$1 - (m-1)/m * (m-2)/m * ... * (m-n+1)/m$$

44.
```
int size(node * hashtable[])
{
     int size = 0;
     int i;
     node * start;
     for (i = 0; i < LENGTH; i++)
     {
          start = hashtable[i];
          while (start != NULL)
          {
               size++;
               start = start-> next;
          }
     }
     return size;
}
```

45. All the strings hash to the same value. For instance, `hash(n) = 4`.

46. Optimization of hash tables, such as by altering its size or by using different hash functions, can yield dramatically different results.

47. It takes up too much memory.

48.
```
0:
1:
2: peach
3:
4:
5:
6:
7: kiwi->apple->Cherry
8: lemon->banana
9: clementine
```

49. There are potential problems with using `scanf`, `strcpy`, and the third and fifth `printf`s. One better way to write this would be:

```
#include <stdio.h>
int main(int argc, char **argv) {
     char buf[256], buf2[256];
     printf("Please enter a string\n");
     scanf("%255s", buf);
     strncpy(buf2, argv[1], 255);
     printf("The first command line argument you supplied was %s\n", buf2);
     printf(" the string you entered was %s", buf);
     return 0;
}
```

50. `<imgsrc="blueflax.jpg" alt="Blue Fax (Linumlewisii)" width=300 height=175 align="left" />`

51. c

52. (a) `SELECT password FROM users WHERE username='malan'`
    (b) `UPDATE users SET password='n3rd' WHERE username='malan'`

53.
```
<? $result = mysql_query("SELECT * FROM users"); ?>
<tr><td>username</td><td>password</td><td>fullname</td></tr>
<? while ($row = mysql_fetch_array($result)) { ?>
<tr>
<td><? print($row["username"]) ?></td>
<td><? print($row["password"]) ?></td>
<td><? print($row["fullname"]) ?></td>
</tr>
<? } ?>
```

54. (a) 600
    (b) 650
    (c) 651
    (d) `rw--xr-x`

55. Packets

56. `$password = mysql_real_escape_string($_POST["password"]);`

57. Change type to type="password"