

## 15-122 : Principles of Imperative Computation, Spring 2014

## Written Homework 10

Due before class: Thursday, April 3, 2014

Name: 苟桂霖Andrew ID: W201514613~~Recitation:~~ ACM 1501class

The written portion of this week's homework will give you some practice working with C programming issues and AVL trees. You can either type up your solutions or write them *neatly* by hand, and you should submit your work in class on the due date just before lecture begins. Please remember to *staple* your written homework before submission.

Question	Points	Score
1	4	
2	4	
3	7	
Total:	15	

You must do this assignment in one of two ways and bring the stapled printout to the handin box on Thursday:

- 1) Write your answers *neatly* on a printout of this PDF.
- 2) Use the TeX template at <http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15122-s14/www/theory10.tgz>

## 1. C Program Behavior

For each of the following problems, state what is wrong with the code in one sentence. Do not just try to compile it and write down the error message. (Some of these will compile without error, and some will even run and produce output, but they all contain conceptual errors that may affect correctness.) Read the code and explain what is being done wrong, conceptually.

- (1) (a) 

```
#include <stdio.h>
#include <string.h>
int main() {
    ...
    char *w;
    strcpy(w, "C programming");    // copy string to w
    printf("%s\n", w);
    ...
    return 0;
}
```

Solution: the pointer 'w' is dangling, when strcpy to it, it may override arbitrary memory location, which may results in unexpected behavior or even a segfault.

- (1) (b) 

```
#include <stdio.h>
#define MULT(X,Y) (X*Y)
int main() {
    ...
    int c = MULT(2+3, 3+4);
    printf("(2+3)*(3+4) is = %d\n", c);
    ...
    return 0;
}
```

Solution:  $MULT(2+3, 3+4)$  will translate to  $2+3 \times 3+4$ , so 15 will be printed.

- (1) (c) 

```
#include <stdlib.h>
#include "xalloc.h"
int main() {
    int *a = xmalloc(100);
    for (int i=0; i<100; i++)
        a[i]=i;
    ...
    free(a);
    return 0;
}
```

**Solution:**  
 Xmalloc will alloc 100 bytes, but the for loop requires 100x4 bytes. So the later 300 bytes (75 numbers) will be written to unallocated memory, which may override other data or result in a seg fault.

- (1) (d) This code fragment shows a C function that is called from another function. It is supposed to return the result only if no overflow occurs. *Hint:* You might want to read the section on integers in the C0 to C tutorial <http://c0.typesafety.net/tutorial/From-C0-to-C:-Basics.html#wiki-integers>.

```
#include <assert.h>
int oadd(int x, int y) {
    int result = x + y;
    if (x > 0 && y > 0) assert(result > 0);
    if (x < 0 && y < 0) assert(result < 0);
    return result;
}
```

**Solution:**  
 Seems correct, regardless of the overflow of  $x+y$  is undefined behavior.

## 2. Pass by reference using C

At various points in our C0 programming experience we had to use somewhat awkward workarounds to deal with *functions that need to return more than one value*. The address-of operator (&) in C gives us a new way of dealing with this issue. In C, the expression &x evaluates to the address of the variable x.

- (2) (a) Sometimes, a function needs to be able to both 1) signal whether it can return a result, and 2) return that result if it is able to. Consider the following function `parse_string` that parses a string into an integer if it is possible:

```
bool parse(char *s, int *i);
// Returns true iff parse succeeds

void parse_string(char *s) {
    REQUIRES(s != NULL);
    int *i = xmalloc(sizeof(int));
    if (parse(s, i))
        printf("Success: %d.\n", *i);
    else
        printf("Failure.\n");
    free(i);
    return;
}
```

The `parse_string` function relies on `parse` which both sets `*i` to an integer equivalent to the integer pattern in `*s` (if possible) and also returns a boolean value of true if the parse succeeds, or false otherwise.

Using the address-of operator, rewrite the body of the `parse_string` function so that it does not heap-allocate, free, or leak any memory on the heap. You may assume `parse` has been implemented (its prototype is given above).

### Solution:

```
void parse_string(char *s) {
    REQUIRES(s != NULL);
    int i;
    if (parse(s, &i))
        printf("Success: %d.\n", i);
    else
        printf("Failure.\n");

    return;
}
```

- (2) (b) In both C and C0, multiple values can be ‘returned’ by bundling them in a struct:

```
struct bundle { int x; int y; };
struct bundle *foo(int p) {
    ...
    struct bundle *A = xmalloc(sizeof(struct bundle));
    A->x = e1;      // first value to be returned
    A->y = e2;      // second value to be returned
    return A;      // return both values together as a struct
}
int main() {
    ...
    struct bundle *B = foo(p);
    int x = B->x;
    int y = B->y;
    free(B);
    ...
}
```

Rewrite the declaration and the last few lines of the function `foo`, as well as the snippet of `main`, to avoid heap-allocating, freeing, or leaking any memory on the heap. The rest of the code (...) should continue to behave exactly as it did before.

**Solution:**

```
void foo(struct bundle *A, int p) {
    ...
    A->x = e1;
    A->y = e2;
    return;
}

int main() {
    ...
    struct bundle B;

    foo(struct bundle *A, p);

    int x = B->x;
    int y = B->y;
    ...
}
```

## 3. AVL Trees.

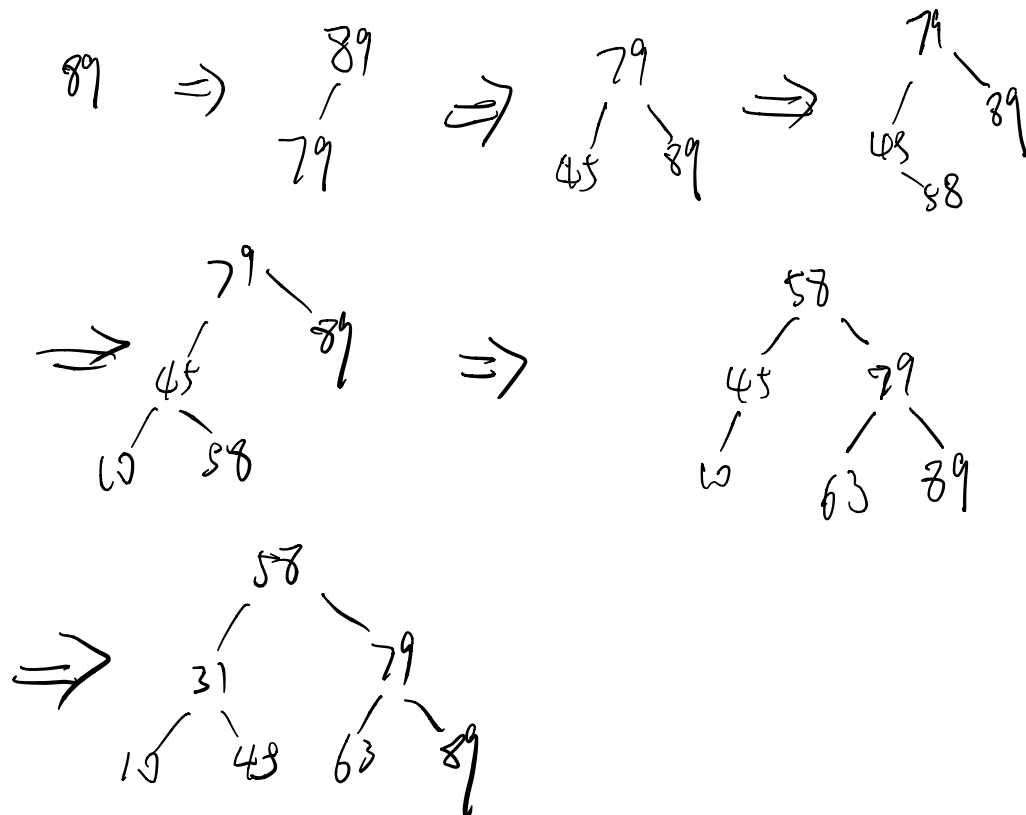
- (3) (a) Draw the AVL trees that result after successively inserting the following keys into an initially empty tree, in the order shown:

89, 79, 45, 58, 10, 63, 31

Show the tree after each insertion and subsequent re-balancing (if any) is completed: the tree after the first element, 89, is inserted into an empty tree, then the tree after 79 is inserted into the first tree, and so on for a total of seven trees. Make it clear what order the trees are in.

Be sure to maintain and restore the BST invariants and the additional balance invariant required for an AVL tree after each insert.

**Solution:**



- (b) Recall our definition for the height  $h$  of a tree:

**The height of a tree is the maximum length of a path from the root to a leaf. So the empty tree has height 0, the tree with one node has height 1, and a balanced tree with three nodes has height 2.**

The minimum number of nodes  $m$  in a valid AVL tree is related to its height. The goal of this question is to quantify this relationship.

- (2) i. Fill in the table below relating the variables  $h$  and  $m$ :

$h$	$m$
0	0
1	1
2	2
3	4
4	7
5	
6	

Handwritten notes next to the table:  
 For  $h=1$  to  $h=4$ , the values of  $m$  are  $1, 2, 4, 7$ .  
 Brackets indicate the recurrence:  $m(h) = m(h-1) + m(h-2) + 1$ .

- (1) ii. Guided by the table in part (i), give an expression for  $m$  as a function of  $h$ . Here's a hint: recall that the  $n$ th Fibonacci number  $F(n)$  is defined by:

$$F(0) = 0$$

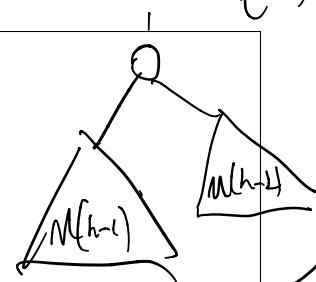
$$F(1) = 1$$

$$F(n) = F(n-1) + F(n-2), \quad n > 1$$

You may find it useful to use the Fibonacci function  $F(n)$  in your answer. Your answer does not need to be a closed form expression; it could be a recursive definition like the one for  $F(n)$ .

Solution:

$$m(h) = m(h-1) + m(h-2) + 1$$



- (1) iii. Give a closed form expression for  $M(h)$ , the *maximum* number of nodes in a valid AVL tree of height  $h$ .

Solution:  $M(h) = F(n+1) - 1 = \frac{\phi^{n+1} - (-\phi)^{n+1}}{\sqrt{5}} - 1$