# C Program Behavior

（1a）

#include <stdio.h>

#include <string.h>

int main() {

...

char \*w;

strcpy(w,"C programming"); // copy string to w

printf("%s\n", w);

...

return 0;

}

**w为野指针，未指向任何内存空间，没有访问权限的地址**

（1b）

#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;

}

**宏只是进行简单的替换， MULT(2 + 3, 3 + 4)展开后变成“2 + 3 \* 3 + 4” ，运算顺序改变，结果为15，与预期的结果35不符；正确的定义为” #define MULT(X, Y) ((X) \* (Y))”**

（1c）

#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;

}

**int类型的字节数为4，只申请了100个字节大小的空间，会造成内存溢出；**

（1d）This code fragment shows a C function that is called from another function. It is

supposed to return the result only if no overflflow 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;

}

**两个负数相加，如果发生溢出result仍然可能小于0**

# Pass by reference using C

（2a）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) != 0)

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 equiv

alent 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).

**void parse\_string( char \*s)**

**{**

**REQUIRES(s != NULL);**

**int i;**

**if(parse(s, &i))**

**printf("Success: %d.\n", i);**

**else**

**printf("Failure.\n");**

**return;**

**}**

（2b）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.

void foo( ***struct bundle \*A*** , int p) {

...

A->x = e1;

A->y = e2;

return;

}

int main() {

...

struct bundle B;

foo( ***&B*** , p);

int x = ***B->x*** ;

int y = ***B->y*** ;

...

}

# AVL Trees

（3a）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:

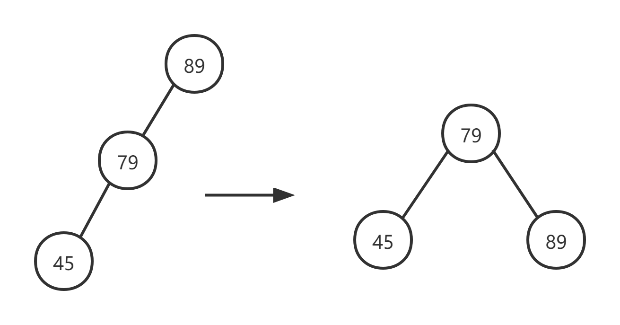
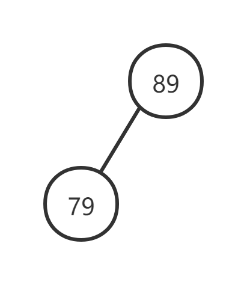
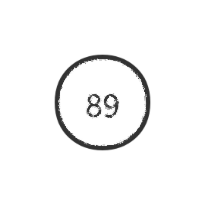
the tree after the fifirst element, 89, is inserted into an empty tree, then the tree after

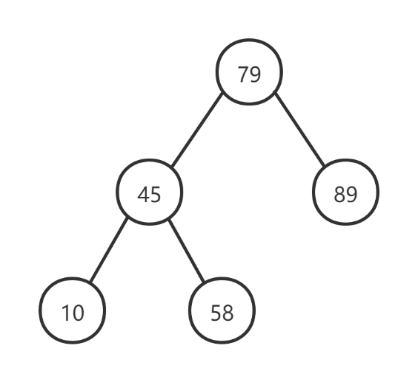
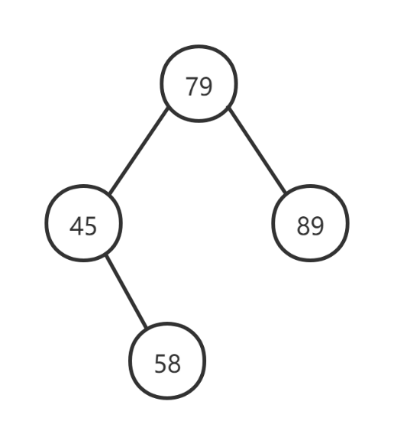
79 is inserted into the fifirst tree, and so on for a total of seven trees. Make it clear

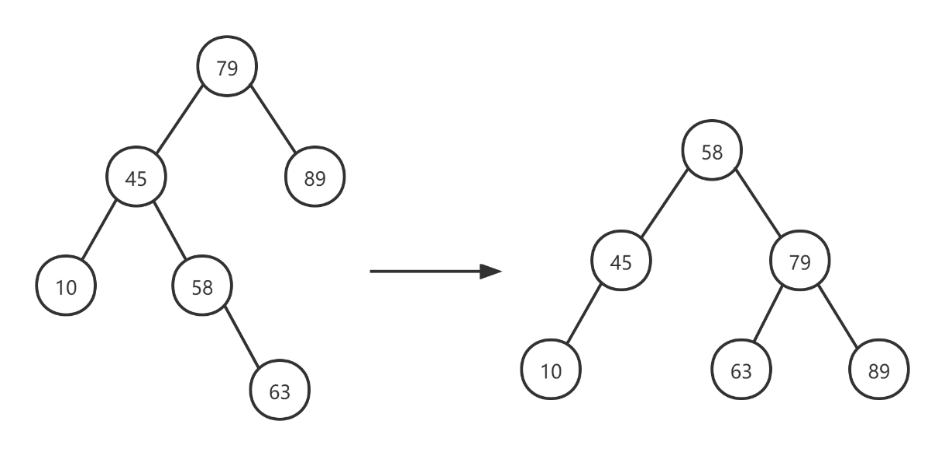
what order the trees are in.

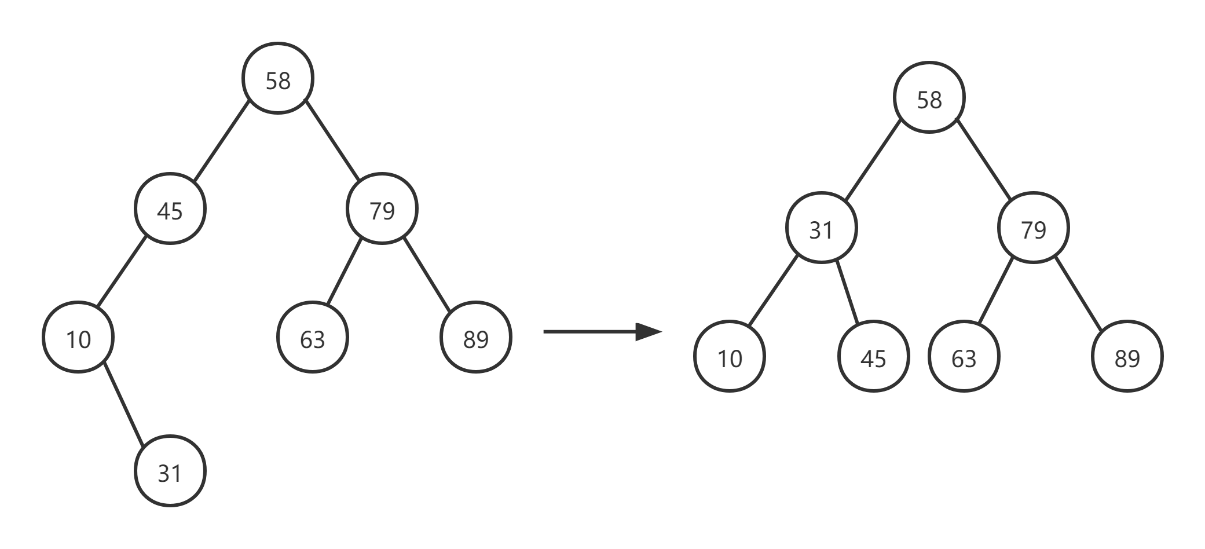
Be sure to maintain and restore the BST invariants and the additional balance

invariant required for an AVL tree after each insert.









（3b）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.

（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** | **12** |
| **6** | **20** |

（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 defifined by:

*F*(0) = 0

*F*(1) = 1

*F*(*n*) = *F*(*n −* 1) + *F*(*n −* 2)*, n >* 1

You may fifind 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

defifinition like the one for *F*(*n*).

**F(h) = F(h-1) + F(h-2) +1。**

（iii）Give a closed form expression for *M*(*h*), the *maximum* number of nodes in a

valid AVL tree of height *h*.

**M(h) = – 1.**