For problems A-D, submit C programs producer.c, producer.h, scheduler.c, scheduler.h, B.c, B.h, C.c, C.h, D.c, D.h, E.c and E.h. Create a makefile with the following targets:

- **producer**: Build producer using producer.c and producer.h with runtime assertions inactive and no debugging information included in the image.
- **scheduler**: Build scheduler using scheduler.c and scheduler.h with runtime assertions inactive and no debugging information included in the image.
- **debug_producer**: Build producer using producer.c and producer.h with runtime assertions active and debugging information included in the image.
- **debug_scheduler**: Build scheduler using scheduler.c and scheduler.h with runtime assertions active and debugging information included in the image.
- **B**: Build B using B.c and B.h with runtime assertions inactive and no debugging information included in the image.
- **debugB**: Build scheduler using B.c and B.h with runtime assertions active and debugging information included in the image.
- **C**: Build B using C.c and C.h with runtime assertions inactive and no debugging information included in the image.
- **debugC**: Build scheduler using C.c and C.h with runtime assertions active and debugging information included in the image.
- **D**: Build B using D.c and D.h with runtime assertions inactive and no debugging information included in the image.
- **debugD**: Build scheduler using D.c and D.h with runtime assertions active and debugging information included in the image.

When you are finished, use gtar to tar and zip the files and submit via the course turnin webpage.

**Expectations with Regard to Assertions**

Each of your functions should begin with assertions that would stop a co-worker from calling your function with invalid parameters, or invalid states of global variables. For example, if your function takes a pointer, you must check if it is NULL, unless you have behavior that your function performs if given a NULL pointer. If a number cannot be zero, you must place an assertion to that effect. I expect you to recognize the difference between an assertion (the function is called in clear violation of its documented range) and handling strange values (for example, the user types the wrong letter).
A. Write two C programs called producer and scheduler. The producer should loop indefinitely until it receives a SIGINT or SIGTERM. When it receives a SIGUSR1, it should print “Producing n” where n begins at 1 and is incremented each time the message is printed. The scheduler program should take a parameter representing the process ID of the producer process. The scheduler should present the user with a menu that has three options:

- Request_Production
- Stop_Producer
- Stop_Scheduler

In order to prevent excessive drain on ix, a shared resource, use a loop in your producer program that sleeps for 2 seconds at a time, using the sleep() function defined in unistd.h.

B. Write a C program uses the state pattern to simulate the Deschutes Hall Pop Machine. The input can consist of lines containing one of the following:

- quarter
- dollar
- pepsi
- diet
- dew
- mist
- water
- pepper
- exit

The output of your program may consist of lines containing the following output:

- accept quarter
- accept dollar
- reject quarter
- reject dollar
- dispense quarter
- dispense pepsi
- dispense diet
- dispense water
- dispense mist
- dispense pepper
- shutting down

The input represents the customer either depositing a quarter or a dollar or pressing a product button, and the output represents the machine’s action for each input. Bottled water costs $1.00. Every other product costs $1.50. The machine will reject any money input once it has reached $1.50. When a product is input, if the user has input at least the cost, the machine will first dispense the product, then it will dispense enough quarters to refund any change owed the customer (in that order!) “Exit” in the input means that the machine is unplugged. It will not refund any money when it is unplugged. Your program should represent the FSM on the following page. Each state represents how much money is in the machine, with the negative meaning the machine has that amount of money, but owes it to the customer. The green arrows represent the machine accepting a quarter; the orange arrows represent the machine accepting a dollar; The red arrows represent the machine dispensing some number of quarters; the blue arrow representing the machine dispensing a water, and the black arrows represent the machine dispensing some product other than water. Rejecting a quarter or dollar is a self-loop to the same state, and not shown in the FSM. Pushing a product button without enough money in the machine is also a self-loop, but generates no output.
Each state in the FSM should be represented in your program by a function, which takes the user input, possibly prints some output, and sets a global variable to a function pointer to the next state. Here is some code for `B.h` to get you started:

```cpp
#include <string>

typedef void (*state)(string);
void have0(string);
void have25(string);
void have50(string);
void have75(string);
void have100(string);
void have125(string);
void have150(string);
void have175(string);
void have200(string);
void have225(string);
void owe125(string);
void owe100(string);
void owe75(string);
void owe50(string);
void owe25(string);
state current_state = have0;
```
C. Write a C program that reads numbers from standard in until it reads a zero. For each input number \( n \), your program should output any of the following messages on a line by itself on standard out:

- \( n \) is probably prime.
- \( n \) is divisible by two.
- \( n \) is divisible by three.
- \( n \) is divisible by five.
- \( n \) is not an Euler prime.

Your program should accept the following switches:

- `-w` try dividing the number by 2 and testing Euler primacy using \( p = 2 \).
- `-h` try dividing the number by 3 and testing Euler primacy using \( p = 3 \).
- `-f` try dividing the number by 5 and testing Euler primacy using \( p = 5 \).

Recall that \( n \) is an Euler number with respect to \( p \) if \( p^{(n-1)/2} \equiv \pm 1 \pmod{n} \) for some prime \( p \). \( n \) is an Euler prime, if it is also prime, otherwise, it is an Euler pseudoprime. Most Euler numbers are Euler primes. Depending on the switches, your program should conduct tests and output the appropriate lines. Once a number is found to be composite, your program should stop testing that number. If a number passes all tests indicated by the switches, then it should print “\( n \) is probably prime.” If the `-w` switch is specified, then 2 should be used to test first. If the `-h` switch is specified, the tests for 3 should be conducted before any tests using 5.

Your program should not make use of the imprecise C `pow()` function. Rather, recall that we can use the following identity to quickly calculate powers:

\[
    a^b = \begin{cases} 
        (a^{b/2})^2 & \text{if } b \text{ even} \\
        a(a^{b-1}) & \text{if } b \text{ odd}
    \end{cases}
\]

You should implement a recursive function using this identity. Also, recall that

\[
    ab \mod q = (a \mod q)b
\]
D. Write a C program that creates an unbalanced binary search tree structure. Your program should read positive numbers from standard in until it reads a zero. Then, your program should output the tree structure. The first number read will be the root of the tree, and subsequent numbers should be put in the correct place. That is sufficient since your tree is not self-balancing. Recall that if we add 5 to the binary search tree on the left, it will look like the tree on the right.

Your output should represent a pre-order walk with null values included, contain each node value on a single line and represent each change in depth by prepending a tab to the line. Each node should be represented by the following data structure:

```c
struct treenode{struct treenode* left; struct treenode* right; int value;};
```

Hint: Use a typedef so you don’t have to type `struct` all the time.

<table>
<thead>
<tr>
<th>Sample Input</th>
<th>Sample output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>5</td>
<td>NULL</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>NULL</td>
</tr>
</tbody>
</table>