Assignment 4

Instructions: Write a makefile with targets A, B, C, D clear, debugA, debugB, debugC and debugD. (Hint: your makefile should contain additional targets that I won’t use directly.) The clear target should delete any executable files and object files created by other targets. The A target should create the A problem program without any debugging code. Likewise for the B, C and D targets. The debugA , debugB, debugC and debugD targets should create the respective programs with full debugging code for the gdb debugger. For each problem, write as many source files as you need and name them appropriately, but make sure that the makefile correctly compiles each problem. Each C++ class that you write should have its own header and code file (except the state class in C and D, which should have only a header file). Tar and zip your solutions and submit them via the turn-in script.

A) Write a C++ program that creates an unbalanced binary search tree structure. Your program should read directives from standard in until it reads an exit directive. The following are legal directives:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit</td>
<td>Quit the program</td>
</tr>
<tr>
<td>insert x</td>
<td>Insert a new node with the int value x into the correct position. If a node already exists with the value x, this does nothing.</td>
</tr>
<tr>
<td>delete x</td>
<td>Find a node containing the int value x and delete the node. If no node contains x, this does nothing.</td>
</tr>
<tr>
<td>test x</td>
<td>Print “found x” if any node contains the int value x; otherwise, print “x not found”.</td>
</tr>
<tr>
<td>print inorder</td>
<td>Do an inorder walk of the tree and print the non-null values on a single line with a space between each</td>
</tr>
<tr>
<td>print preorder</td>
<td>Do a preorder walk of the tree and print the values as in problem C of assignment 3.</td>
</tr>
<tr>
<td>print tree</td>
<td>Print the same as print preorder, but when a node has two NULL children, don’t print the NULLs.</td>
</tr>
</tbody>
</table>

Illegal directives should be ignored. When deleting nodes, make sure you free the memory so as not to create a memory leak. Recall that if we want to delete a node:

If both children are null, we delete the node.
If one child is null, we put the non-null child in the parent’s place
If neither child is null, we swap values with the node to be deleted and the rightmost leaf of the left subtree, and then delete the leaf we just made redundant. Thus, if we want to delete the 6 from the tree on the left, we make the 6 a 5, and delete the node that originally contained the 5.
Each node should be represented by a class called treenode.

<table>
<thead>
<tr>
<th>Sample Input</th>
<th>Sample output</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert 2</td>
<td>-1 2 4 5 6 7 8</td>
</tr>
<tr>
<td>insert 8</td>
<td>6 not found</td>
</tr>
<tr>
<td>insert -1</td>
<td>2</td>
</tr>
<tr>
<td>insert 6</td>
<td>NULL</td>
</tr>
<tr>
<td>insert 4</td>
<td>0</td>
</tr>
<tr>
<td>insert 5</td>
<td>8</td>
</tr>
<tr>
<td>print inorder</td>
<td>5</td>
</tr>
<tr>
<td>delete 6</td>
<td>4</td>
</tr>
<tr>
<td>insert 0</td>
<td>7</td>
</tr>
<tr>
<td>test 6</td>
<td>NULL</td>
</tr>
<tr>
<td>print tree</td>
<td></td>
</tr>
</tbody>
</table>

**B)** The following BNF grammar describes a prefix arithmetic problem on natural numbers.

\[
E ::= \text{unsignedInt space} | \text{operator} E E
\]

The production \textit{unsignedInt} is given by the regular expression \([0-9]+\) and \textit{operator} is given by the regular expression \([-*/^]\). Note that this definition requires a trailing space character.

Write a C++ program that reads from the standard input stream and prints the infix version of the input to the standard output stream. Thus, if the input is \(\ast\-2\ 3\ 4\), the output would be \(2-3\ast4\). In infix, there should be no spaces, but the expression must be properly parenthesized. To properly parenthesize a subtree, look at the node above the current node. If the parent has greater precedence than the child, then the child’s subtree has to be put in parentheses.

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>(^)</td>
</tr>
<tr>
<td>2</td>
<td>(*\ /)</td>
</tr>
<tr>
<td>1</td>
<td>(+\ -)</td>
</tr>
</tbody>
</table>

Your program should continue to read lines until it encounters a line containing only zero. Your program should stop when it reads a line containing only zero.
C) Write a C++ program to simulate the Deschutes Hall Pop Machine. The input can consist of lines containing one of the following:

- quarter
- dollar
- pepsi
- diet
- dew
- mist
- pepper
- water
- 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 dew
- dispense mist
- dispense pepper
- dispense water
- 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 class, which has a member function that takes the user input, possibly prints some output, and returns the next state. Here is some code to get you started (break into the appropriate files, though):
```cpp
#include <iostream>
#include <string>
using namespace std;

class state {
    public: state* transition(string input) {return this;}
};

class have0: public state {
    public: state* transition(string input);
};

// more class declarations here
State* have0::transition(string input) {
    if (input == "dollar")
        return new have100();
    if (input == "quarter")
        return new have25();
    return this;
    //test in: take a quarter or dollar, ignore others.
    //Return next state accordingly.
}

// more class definitions here
int main(int argc, char** argv) {
    string input; state* current_state = new have0();
    //get input
    while (input != "exit") {
        current_state = current_state->transition(input);
        //get input
    }
}

State diagram for Pop Machine FSM
```
D) In part C, you feel that you are doing way too much creation of tiny, short-lived objects, leading to thrashing behavior in the heap. You notice that your states are stateless. That is, the classes that you use in the state pattern have no internal variables, only non-virtual member functions. That means that you can implement the singleton pattern as well. The singleton pattern says that for each state subclass, you declare the constructor to be private and a single instance of the class to be public and static in the class itself. This “singleton” object is the only copy of that class that is allowed to exist. For this problem, retrofit your answer to part C to implement the singleton pattern for all the state objects (except state, the superclass). The input and output behavior of your program should be exactly the same as in part C, but there will not be as many heap allocations.