CMSC 212 Discussion exercise, Monday, November 16 Fall 2009

Suppose a debugger for a small language needs to store information about the variables which are used by a program. This particular language only has `int` and `char` variables. Every variable has some common information which always needs to be stored– its name and the name of the function in which it was declared. Besides that, an `int` variable also needs to have an `int` value stored (the variable's current value), and a `char` variable needs to have a `char` value stored (for its value).

We've given you a program which reads variables and stores their data in a binary search tree. Each `Node` in the tree contains a variable's name, the name of the function it's declared in, an `enum` indicating what type of variable it is, a `union` storing its value (either an `int` or a `char`), and left and right pointers to the subtrees of the node. Compile the code with the `Makefile` given and run it with input redirected from the file `input`, as in the command `main.x < input`, and you'll see the data for the variables printed out in alphabetical order by variable name, since the variables are inserted into the tree according to their name (variables' names are used as the search key).

What you are to do is to modify the program so that variables can be stored in the tree either in order by their names, or in order by their values, using function pointers. In other words, the search key used to look up or insert variables will be either their name, or their value (its numeric value if it's an integer variable, or its ASCII value if it's a character variable). A pointer to a function will now be stored in each node which can be called to get the key of the type of variable stored in that node, and another function pointer will be stored in each node which is for a function which prints the value of the variable stored in that node.

If you don't finish this exercise in discussion you should keep on working on it, as it's important practice for material needed later. But even if you don't finish, don't forget to submit what you've done by the end of your discussion section, whether it works right or not, to get credit for the exercise.

1. You'll find a compressed tarfile `discussion22.tgz` in our directory `212public/discussions` which you created a symbolic link to. Copy it to your `212` directory and `cd` there.

2. Extract the files using `gtar -zxvf discussion22.tgz`, which will create a subdirectory `discussion22`, so `cd` there. You can compile the program and run it with the command above, to see the variables being printed in order by name, since they're stored in the tree according to the values of their names.

3. In the header file `variant-tree.h`, uncomment the definitions of `Get_key_function`, `Print_function`, and `Compare_function`. The first two are function pointers which have a `Value` union as a parameter; `Get_key_function` returns an int and `Print_function` has no return value. `Compare_function`, which is a pointer to a function which takes two `Node`s as parameters and returns an `int`.

4. Add two fields to the `Node` structure– one of type `Get_key_function` and another one of type `Print_function`.

5. In `variant-tree.h` and `variant-tree.c`, add another parameter to `insert()`, of type `Compare_function`, which will instead be called to compare `Node`s when inserting. Right now, the `insert` function just inserts variables using their names as a key, but we want it to use a function which will be passed into `insert()` instead.

6. Add prototypes in `variant-tree.h` for two functions named `get_int_key()` and `get_char_key()`, which have a `Value` parameter and return an `int`, and write them in `variant-tree.c`. The first one just returns the `int` value of the parameter (in the union), while the second one just returns the `char` value.

7. Now add prototypes for and write two one–line functions `print_int()` and `print_char()`, which have a `Value` parameter and no return value. The first one just prints the `int` value of the parameter (in the union), while the second one just prints the `char` value.

8. `insert()` is called from `create_tree()`. Where the new node's fields are being filled in in `create_tree()`, depending upon the value of the third field on each line of the input (whether it's the string "`int`" or the string "`char`"), add statements to store the correct two functions in the function pointer fields (which you added to the `Node` structure) of the new `Node`. If the variable is an int, the node's function pointers should point to `get_int_key` and `print_int`. If it's a char, its function pointers should point to `get_char_key` and `print_char`.

9. `create_tree()` is called from `main()`. Add a parameter to `create_tree()` of type `Compare_function`, which `main()` will use to indicate whether we want to store and print values in the tree in order by their names, or by their values. Then `create_tree()` will have to pass this function pointer along to `insert()`.

10. Write two functions `compare_var_names()` and `compare_var_values()`, which `main()` will pass via this parameter into `create_tree()`. They have two `Node`s as parameters and return an `int`.

    - `compare_var_names()` should compare the names of the variables in its two `Node` parameters, returning either -1, 0, or 1, to indicate whether the name of the first `Node` is less than, equal to, or greater than the second one (the function can just call `strcmp()` for this.
    - `compare_var_values()` should compare the values of the variables in its two `Node` parameters, returning either -1, 0, or 1, to indicate whether the value of the first `Node` is less than, equal to, or greater than the second one. It should just call `get_key()` on its two parameters, and compare the results, to decide whether to return -1, 0, or 1.

    You can pass one of these two functions as the second parameter to `create_tree()` where it's called from `main()` (whichever one you want). If you call `compare_var_names()` the program will insert variables in the tree in order by their names, while if you calll `compare_var_values()` it'll insert them in order by their values. Go ahead and call `compare_var_values()` for now, so you can see the program produce different output than it did before.

11. Lastly, modify `print_tree` so that instead of printing the `int_value` or `char_value` field of the variable in each node, it just calls the `Print_function` function pointer stored in each node, passing it the `value` field in that node, to print the type of data in that node.

The result of all of the above is if you passed a pointer to `compare_var_values` from `main()` to `create_tree()`, the program will store variables in the tree in order by their values, and give the output on the left below, while if you passed a pointer to `compare_var_names` from `main()` to `create_tree()` the program will store variables in the tree in order by their names and give the output on the right.

```
Variable name is "index", its value is 10.
Variable name is "arr", its value is 27.
Variable name is "sub", its value is 'a'.
Variable name is "y", its value is 'z'.
Variable name is "register", its value is 1000.
```

```
Variable name is "arr", its value is 27.
Variable name is "index", its value is 10.
Variable name is "register", its value is 1000.
Variable name is "sub", its value is 'a'.
Variable name is "y", its value is 'z'.
```