SEU-ICS : C Programming lab
Stack

Overview

This lab will give you practice in the style of programming you will need to be able to do proficiently, especially for the later assignments in the class. The material covered should all be reviewed for you. Some of the skills tested are:

- Explicit memory management, as required in C.
- Creating and manipulating pointer-based data structures.
- Implementing robust code that operates correctly with invalid arguments, including **NULL pointers**.

The lab involves implementing a stack, supporting first-in, last-out (FILO) disciplines. The underlying data structure contains two pointers: (1) *bottom to point to the bottom of the stack and (2) *top pointing to the top of the stack. An additional variable "int cnt" is used to represent the factual size of the stack, which increases when an element is pushed into the stack.

- s_new: Create a new, empty stack.
- s_free: Free all storage used by a stack.
- s_push: Attempt to insert a new element at the top of the stack.
- s_pop: Attempt to remove the element at the top of the stack.
- s_size: Compute the number of elements in the stack.
- s_empty: Determine whether the stack is empty.
- s_reverse: Reorder the list so that the stack elements are reversed in order.

Assignment

Your lab materials are contained in an archive file called stacklab .zip. The file stack.h contains declarations of the following structures:

```
/* Linked list element (You shouldn't need to change this) */
typedef struct node{
    int value;
    struct node *next;
} s_node;

/* Stack structure */
typedef struct {
    s_node *top;  /* Linked list of elements */
    s_node *bottom;
    int cnt;
} stack;
```

These are combined to implement a stack. The top-level representation of a stack is a structure of type stack. In the starter code, this structure contains the "top", "bottom", and "cnt", but you will want to add other fields. The stack contents are represented as a singly-linked list, with each element represented by a structure of type s_node, having fields “value” and “next,” storing a stack value and a pointer to the next list element, respectively. In our C code, a stack is a pointer of type stack *. We distinguish two special cases: a NULL stack is one for which the pointer is set to NULL. An empty stack is one pointing to a valid stack structure with the bottom set to be the top field. Your code will need to deal properly with both of these cases, as well as stacks containing one or more elements.

Testing

You can compile your code using the command:

```
linux> make
```

If there are no errors, the compiler will generate an executable program stest, providing a command interface with which you can create, modify, and examine stacks. Documentation on the available commands can be found by starting this program and running the help command:

```
cmd>help
Commands:
    #         ...             | Display comment
    free                      | Delete stack
    help                      | Show documentation
    log       file            | Copy output to file
    new                       | Create new stack
    option    [name val]      | Display or set options
    pop       [v]             | Remove from top of stack.  Optionally compare to exp
    push      v [n]           | Insert v at top of stack n times (default: n == 1)
    quit                      | Exit program
    reverse                   | Reverse stack
    show                      | Show stack contents
    size      [n]             | Compute stack size n times (default: n == 1)
    source    file            | Read commands from source file
    time      cmd arg ...     | Time command execution
Options:
    echo    1       Do/don't echo commands
    error   5       Number of errors until exit
    fail    30      Number of times allow stack operations to return false
    malloc  0       Malloc failure probability percent
    verbose 4       Verbosity level
```

You are recommended to use new, pop, push, reverse, show, and size to test your code. Here is an example:

```
cmd>new
s = []
cmd>push 2 2
cmd>push 2 2
s = [2 2]
cmd>push 3
cmd>push 3
s = [3 2 2]
cmd>push 199
cmd>push 199
s = [199 3 2 2]
cmd>pop
cmd>pop
```

```
s = [3 2 2]
cmd>pop 3
cmd>pop 3
s = [2 2]
cmd>reverse
cmd>reverse
s = [2 2]
cmd>push 5
cmd>push 5
s = [5 2 2]
cmd>reverse
cmd>reverse
s = [2 2 5]
cmd>size
cmd>size
Stack size = 3
s = [2 2 5]
cmd>free
cmd>free
s = NULL
```

You can see the effect of these commands by operating stest in batch mode:

```
linux> ./stest -f traces/trace-eg.cmd
cmd># Demonstration of stack testing framework
cmd># Use help command to see list of commands and options
cmd># Initial stack is NULL.
cmd>show
s = NULL
cmd># Create empty stack
cmd>new
s = []
cmd># Fill it with some values.  Using push
cmd>push 2
s = [2]
cmd>push 1
s = [1 2]
cmd>push 3
s = [3 1 2]
cmd>push 5
s = [5 3 1 2]
cmd>push 1
s = [1 5 3 1 2]
cmd># Reverse it
cmd>reverse
s = [2 1 3 5 1]
cmd># See how long it is
cmd>size
Stack size = 5
s = [2 1 3 5 1]
cmd># Delete stack.  Goes back to a NULL stack.
cmd>free
s = NULL
cmd># Exit program
cmd>quit
```

```
Freeing stack
```

With the starter code, you will see that many of these operations are not implemented properly. The traces directory contains 14 trace files, with names of the form trace-k-cat.txt, where k is the trace number, and cat specifies the category of properties being tested. Each trace consists of a sequence of commands, similar to those shown above. They test different aspects of the correctness, robustness, and performance of your program. You can use these, your own trace files, and direct interactions with stest to test and debug your program.

Evaluation

Your program will be evaluated using the fourteen traces described above. You will given credit (either 7 or 8 points, depending on the trace) for each one that executes correctly, summing to a maximum score of 100. The driver program driver.py runs stest on the traces and computes the score. This is the same program that will be used to compute your score. You can invoke it directly with the command:

```
linux> ./driver.py
```

or with the command:

```
linux> make test
```