UMEÅ UNIVERSITY **April 21, 2020**
Department of Computing Science
Assignment 2 report

5DV149 - Assignment 2, v1.0

Data Structures and Algorithms (C) Spring 2020, 7.5 Credits

Test of stack implementation

Name	Elias Olofsson
user@cs.umu.se	`tfy17eon@cs.umu.se`

Teacher
Anna Jonsson

Graders
Anna Jonsson, Lena Kallin Westin, Hannah Devinney

Contents

1 Description of interface

In this lab we have designed and implemented a test program to verify the functionality of a given implementation of the abstract datatype *stack* in the programming language C. The interface of the datatype as implemented in `stack.c` is given below.

```
implemented datatype stack.c
   stack_empty(free_function free_func) → stack *
   stack_is_empty(const stack *s) → bool
   stack_push(stack *s, void *v) → stack *
   stack_pop(stack *s) → stack *
   stack_top(const stack *s) → void *
   stack_kill(stack *s) → void
   stack_print(const stack *s, inspect_callback print_func) → void
```

The first function in the interface of the stack implementation is the `stack_empty` function which creates and returns a pointer to a new and empty stack. As an argument, it takes a pointer `free_func` to a function which will be called at the time of memory deallocation and removal of the stack elements. If no freeing function is specified, memory deallocation must be handled manually by the user for the element values. Since the datatype stack is homogeneous and of poly-type, it can accept element values of any datatype, but for each specific instance of stack, only one element datatype at a time is allowed.

The second function `stack_is_empty` takes a pointer `s` to a stack and checks whether the stack is empty. The function returns true if the stack is in fact empty.

The third function `stack_push` takes a pointer `v` to a value of unspecified type, and a pointer `s` to a stack containing values of the same type as `v`, and puts the new value at the top of the stack. Finally, a pointer to the altered stack with the new element on top, is returned by the function.

Next up, the forth function `stack_pop` takes a pointer `s` to a stack as argument, removes the top element of that stack and returns a pointer to the altered stack, now with one less element in it. If a freeing function was specified at the creation of the stack, this is called and memory for the removed element value is freed. The function `stack_pop` is undefined for an empty stack.

The fifth function `stack_top` accepts a pointer `s` to a stack as argument, inspects the element at the top of the stack and returns a pointer to the value of the top element. This function is undefined for an empty stack.

The sixth function `stack_kill` takes a pointer `s` to a stack and destroys the given stack. If a freeing function was specified at the creation of the stack, this is called for memory deallocation of the element values. If no freeing function was specified, manual freeing of allocated memory is the responsibility of the user.

The seventh and last function, `stack_print`, takes a pointer to a stack `s` and prints the element values of the stack in the terminal using the specified printing function `print_func`.

2 Description of tests

Below follows a description of each test performed in `stack_test.c` aimed to verify the functionality of the stack implementation `stack.c`. Each of the test is designed to test one, or parts of one of the five fundamental axioms for the abstract datatype stack. For simplicity are the five logical axioms listed below with their general abstract function names, independent of implementation. The equivalent implemented functions can easily be correlated from the the interface listed above, with exception of `stack_kill` and `stack_print`.

1. `Isempty(Empty()) = true`
2. `¬Isempty(Push(v,s)) = true`
3. `Pop(Push(v,s)) = s`
4. `Top(Push(v,s)) = v`
5. `¬Isempty(s)` $\Rightarrow$ `Push(Top(s),Pop(s)) = s`

2.1 `empty_stack_test()`

This function verifies axiom 1, thus that a newly created stack is empty. A new stack is created using `stack_empty` and thereafter tested with `stack_is_empty`. If the return from `stack_is_empty` is true, the test is passed and memory deallocation is performed. If not, the test has failed and `empty_stack_test` writes out an error message before exiting the program.

Since there is only one way to have an empty stack, there is no other or alternative way to test this axiom. Thus, if the implementation passes this test, it can be considered to fully have met the requirements of axiom 1.

2.2 `nonempty_stack_test1()`

In this function we verify part of axiom 2, thus that a stack is nonempty after having an element pushed to the stack. In particular, we test the delimiting case where a single element is pushed to an empty stack and `stack_is_empty` checks whether if the stack is empty. If the stack is nonempty, the test is passed and memory deallocation occurs. If the stack is empty, the test has failed and an error message is printed before the program exits.

2.3 `nonempty_stack_test2()`

Similarly to `nonempty_stack_test1`, this function also test axiom 2, but for a more general and arbitrary case than the limiting single value case of the previous test. In particular, here a new stack is created and arbitrarily 5 values are pushed in succession to the stack, which thereafter `stack_is_empty` checks if the stack is empty. If the stack is nonempty, the test is passed and memory is freed. If an empty stack is detected, the test has failed and the program exits after printing an error message.

If both `nonempty_stack_test1` and `nonempty_stack_test2` has been passed by the stack implementation, the code could be considered to have fulfilled the requirements of axiom 2. Since the single extreme case for the axiom, plus a complementary arbitrary case have been tested and passed, there is reason to believe that the code is valid in general. Still, if even stronger confidence in the fulfillment of the axiom is required, more complimentary testing is still needed. But as a minimalistic first test, this is enough.

2.4 `one_element_test()`

This function aims to verify parts of axiom 4, thus that a value pushed to a stack can be inspected to be at the top of the stack. In particular, this function tests the delimiting case where a single value is pushed to an empty stack. A known value is pushed to a newly created stack, and thereafter `stack_top` is used to inspect the value at the top of the stack. A comparison between the pushed and the inspected value is performed and the test is passed if the values are equal, whereupon memory deallocation occurs. If the values are not equal, the test has failed and the program exits after printing an error message.

2.5 `multiple_elements_test()`

Similarly to `one_element_test`, this function tests axiom 4. The difference here is that this test is performed for a more general and arbitrary case than the limiting case of the previous test. Here, 5 arbitrary values are pushed in succession to an empty stack, while the value last pushed to the stack is saved in a temporary variable. Thereafter `stack_top` inspects the value at the top of the stack and a comparison with the last pushed value is performed. If equal, the test is passed and memory is deallocated. If the value at the top of the stack is not equal to the value last pushed to the stack, the test is failed and the programs exits after printing an error message.

If both `one_elements_test` and `multiple_elements_test` have been passed by the stack implementation, one can consider the implementation to have met the requirements of axiom 4. Since the single limiting case of the axiom, and an additional arbitrary case have been tested and passed, there is reason to believe that the code is valid in general. Same argument applies here as well though, if stronger confidence is needed, more exhaustive testing is required.

2.6 `empty_push_pop_test()`

In this test the validity of axiom 3 is partly tested. Thus we verify that after pushing a new value to a stack, followed by removing the top value of that same stack, we recover the initial stack. In particular, here we test the limiting case of pushing a single value to an empty stack, and checking that the stack is empty again after popping the top value of the stack. If the stack is empty after the push-pop maneuver, the test is passed and allocated memory is freed. If the stack is not empty, then the stack has changed and the test has failed. The program prints an error message to the terminal and exits.

2.7 `push_pop_test()`

In addition to `empty_push_pop_test`, this function also tests axiom 3, but for a more general and arbitrary case than the limiting case in the previous test. Here, we create two identical but separate stacks by pushing a succession of the same arbitrary values to two newly created stacks. It is important to make the clarification that it is two separate instances of stacks, stored in different bits of memory, that have the same numerical values and internal ordering. Once the two stacks are created and filled up with identical but arbitrary numbers, an additional value is pushed to one of the stacks then promptly removed from the top of the same stack. Following this, we check whether if the two stacks are still identical. In other words, we verify that they still have the same size and have the same element values in the same order. This is performed by repeated application and comparison of `stack_is_empty` and `stack_top` for both of the two stacks, while successively removing and deallocating the top elements until both stacks are empty.

If there is no mismatch in size or element values between the two stacks during the entire duration of this process, the stacks were in fact identical and the test has been passed. The remaining stack structure is deallocated and the function is exited normally. If at any point during the process of comparison the element values happen to differ, or stack is of unequal size, the test has failed and the program exits after printing an error message.

If both `empty_push_pop_test` and `push_pop_test` is passed by the stack implementation, the requirements of axiom 3 can be considered to be fulfilled. Since by then, both the single extreme case of the axiom, plus an additional arbitrary case have been tested and passed, there is reason to believe that the code is valid in the general case.

2.8 `minimal_pop_push_test()`

This function aims to verify parts of the 5:th and final axiom, thus that for a nonempty stack, we recover the initial stack after removing the top element and pushing the same element value back to the same stack. In particular, this code checks the limiting case of having a stack with only one element in it. Here, we create two separate instances of stack and push an arbitrary numerical value to each of the stacks, thus yielding two single-valued, identical but separate stacks. Once this is done, the element value at the top of one of the stacks is retrieved and saved. The top element of the same stack is removed, and the previously saved element value is pushed back to the same stack. Following this, we check whether if the two stacks are still identical, in other words that both stacks still are single-valued and have the same numerical element value. This is performed with repeated checks with `stack_is_empty` and `stack_top` during successive removal of the top element of both stacks until both stacks are empty, same as the procedure in `push_pop_test` seen above.

If there is no mismatch in size or element values between the two stacks, then the two stacks are in fact identical and the test is passed. If not, one of the stacks has been altered somewhere during the pop-push process, and the test has failed. The program exits after printing an error message.

2.9 pop_push_test()

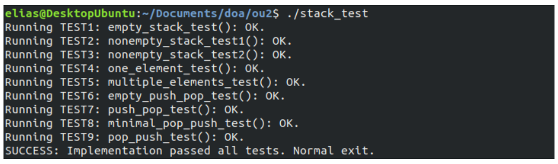
This last function aims to verify axiom 5, same as `minimal_pop_push_test`, but for a more general and arbitrary case. In particular, here we create two new stacks and push five of the same arbitrary numbers in succession to each of the stacks. Thus we have two separate instances of stacks, stored in different bits of memory, but containing identical numerical values and internal ordering. Once this is completed, the top value of one of the stacks is retrieved and saved. The top element of the same stack is removed, after which the saved element value is pushed back to the same stack. Now we proceed to check whether if the two stacks are still identical. This is performed in the same manner as in `push_pop_test` and `minimal_pop_push_test()` above mentioned, thus with repeated checks and comparisons of size and top element values of both stacks during successive removal of the top elements.

If there is no mismatch between stack size and element values throughout this process, the stacks are identical and the test is passed. If any discrepancy is found, one of the stacks have been changed somewhere in the pop-push process, and the test is failed. The program prints an error message and exits.

If both `minimal_pop_push_test` `pop_push_test` is passed by the stack implementation, then one may consider the code to have met the requirements of axiom 5. Since the single limiting case and an additional more arbitrary case have been tested and passed, there is reason to believe that the implementation is valid for the general case.

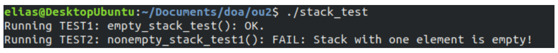
3 Documentation of tests

Below follows the results of the test program `stack_test.c` being ran twice, once for a valid and correct stack implementation, seen in fig.(1), and once for an incorrect implementation where `stack_push` does not do anything, seen in fig.(2).



```
elias@DesktopUbuntu:~/Documents/doa/ou2$ ./stack_test
Running TEST1: empty_stack_test(): OK.
Running TEST2: nonempty_stack_test1(): OK.
Running TEST3: nonempty_stack_test2(): OK.
Running TEST4: one_element_test(): OK.
Running TEST5: multiple_elements_test(): OK.
Running TEST6: empty_push_pop_test(): OK.
Running TEST7: push_pop_test(): OK.
Running TEST8: minimal_pop_push_test(): OK.
Running TEST9: pop_push_test(): OK.
SUCCESS: Implementation passed all tests. Normal exit.
```

Figure 1: Terminal output of `stack_test.c` for an correct stack implementation `stack.c`.



```
elias@DesktopUbuntu:~/Documents/doa/ou2$ ./stack_test
Running TEST1: empty_stack_test(): OK.
Running TEST2: nonempty_stack_test1(): FAIL: Stack with one element is empty!
```

Figure 2: Terminal output of `stack_test.c` for an incorrect stack implementation `stack.c`.