The George Washington University

CSCI 3411 - Operating Systems

Programming Exercise 1 - Linked List

1 Objective

This exercise is designed to reinforce your understanding of C, in particular C pointers, and to implement a data structure that will be critical to your future success in this course. You are to write this implementation from scratch. In order to allocate memory and deallocate memory, you must use `malloc` and `free` from `stdlib.h`. In order to print to the console, you may use `printf` from `stdio.h`. It is also meant to force you to set up a C development environment.

2 Assignment

Implement a stack based linked list in a C file named `list.h` with the following functions:

```
struct linked_list* ll_create(void)

int ll_destroy(struct linked_list *ll)

void ll_add(struct linked_list *ll, void *value)

int ll_length(struct linked_list *ll)

void *ll_remove_first(struct linked_list *ll)

int ll_contains(struct linked_list *ll, void *value)
```

`ll_create` allocates and returns a pointer to a linked list. If unable to allocate the list, `ll_create` returns `NULL`. You must always check if `malloc` returns `NULL`.

`ll_destroy` deallocates a linked list, only if it is empty. Return `1` if the linked list is destroyed, and `0` if it couldn't be destroyed.

`ll_add` inserts a value at the head of the linked list.

`ll_length` returns the total number of values in the linked list.

`ll_remove_first` removes the value at the head of the linked list and returns the value. If the list is empty, `ll_remove_first` returns `NULL`

`ll_contains` searches the linked list from head to tail and returns the first position at which the value is found. In a list with n values, the head is position 1 and the tail is position n; therefore, if the value is in the list, `ll_contains` returns a logical true, the offset into the linked list. If the value is not found in the list, 0 is returned; therefore, if the value is not in the list, `ll_contains` returns a logical false. Most often, this will be used to determine *if* a node is in the linked list, and not where it is in the list, thus the optimization to have it return `0` (logical `false`), and a `1`-indexed offset otherwise (logical `true`).

Your solution should compile with *no errors or warnings* with `gcc -I. -Wall -Wextra -Werror main.c`, and should include no print statements in the header file. We have provided a `Makefile` for this assignment as an example to get you going. We will not provide this in future assignments. Regardless, you must always provide one. You should be able to compile with the command: `make`. `make` must properly compile your program, and generate a `test` binary. You can clean the directory with `make clean`. You will receive no credit if your solution doesn't compile, if doesn't adhere **exactly** to the function prototypes provided (see grading criteria on the webpage). You will receive significant penalties if your solutions compiles with warnings.

3 Testing

You will need to test your `list.h`. Do this by creating a separate file containing a `main` function, called `main.c`. You will need to test each of the functions that you have defined in varying ways.

Consider that each function may behave differently when the list is empty and when the list is non-empty. Too simple a test may give the false impression that the list works properly in all cases when it contains a bug that is only exposed after the list has transitioned through several states. For example, testing the list by only adding values and then removing them fails to consider what happens when a list is emptied and then repopulated.

Additionally, well-structured test data may give the false impression that particular functions work properly. Real-world data is rarely well-structured. Therefore, do not rely on a simple test that uses a set like {1, 2, 3, 4, 5} as the results may be misleading. Your implementation will be evaluated against randomized data.

We will test your program by simply `#include <list.h>`ing your implementation in our test harness. Any part of the specification that you don't test and have working, will likely lead to points off.

4 Submission

Please complete the `list.h` and `main.c` files in your repo. Upon submission, your repo should include *only* `list.h`, `main.c`, `Makefile`, `README.md`, `.gitignore`, and this file. Not adhering to this specification will lead to penalties.
We will test the most recent commit. If that commit is made after the deadline, we will apply late penalties consistent with the classes late policies.

5 Evaluation

You are obligated to complete this exercise on your own by the Code of Academic Integrity (`http://www.gwu.edu/~ntegrity/code.html`).

Your implementation will be compiled against a standard test program that will populate your list with randomized data. If your `list.h` does not match the declarations defined above, your implementation will not compile.

The conciseness and simplicity of your code will affect your evaluation. There is a direct relationship between the probability of introducing bugs and both the number of lines of code implemented and the complexity of individual lines of code. Find a balance between the number of lines of code and the simplicity of each line.

Your code will be evaluated in terms of the comprehensiveness and descriptiveness of comments. Comments are critical to conveying information about your implementation. A block comment preceding your function definition should describe the function including any special information about the input parameters and output result. Line commments should clarify variable usage and complex logic. Do not assume that either you or others will understand code that you have written. The time it takes for anyone to understand code is non-productive.

6 Debugging

There are two primary pointer issues to fundamentally address, segmentation faults and memory leaks. Segmentation faults occur when a pointer is improperly allocated, accessed or deallocated. Memory leaks occur when all pointers are not properly deallocated. Segmentation faults throw exceptions and memory leaks do not. If your code generates a segmentation fault, your evaluation will be affected. Expect your code to be profiled to check for memory leaks. Failing to master pointer allocation and deallocation now will have a drastic impact on your work products and your overall course evaluation.

It is recommended to use a debugger such as `gdb`. In order to use a debugger, you will need to compile with debugging symbols. If compiling with `gcc` from the command line, use the `-g` argument. `gdb` will enable you to set breakpoints and print stack frames so that you can view the state of the program during execution. Do not rely on printing state information to the console.

To compile `list.h` and `main.c` into an executable binary with the name `test`, execute the following command in the directory where these files reside:

```
gcc -Wall -Wextra -g -o test main.c list.h
```

To run `test` under `gdb`, execute the following command in the directory where `test` resides:

```
gdb ./test
```

In `gdb` type:

```
run
```

If a segfault is generated, the program will break at that point and `gdb` will report the exception and wait at a prompt. At a `gdb` prompt, you may print the stack frames by typing:

```
backtrace
```

To check for memory leaks in your test program, you may use a memory profiler such as `valgrind`. `valgrind` will not be emphasized in this course as it is not useful in terms of kernel programming; however, you should use it for this assignment to ensure that you are properly deallocating pointers.

To run `valgrind`, execute the following command in the directory where `test` resides:

```
valgrind ./test
```

7 References

gcc: `http://gcc.gnu.org/onlinedocs`

gdb: `http://www.gnu.org/software/gdb/documentation`

valgrind: `http://valgrind.org/docs/`