15-122: Principles of Imperative Computation Spring 2020

Lab 7: List(en) Up! Tuesday February 25th

Collaboration: In lab, we encourage collaboration and discussion as you work through the problems. These activities, like recitation, are meant to get you to review what we've learned, look at problems from a different perspective and allow you to ask questions about topics you don't understand. We encourage discussing problems with your neighbors as you work through this lab!

Setup: Copy the lab code from our public directory to your private directory:

```
% cd private/15122
% cp -R /afs/andrew/course/15/122/misc/lab07 .
% cd lab07
```

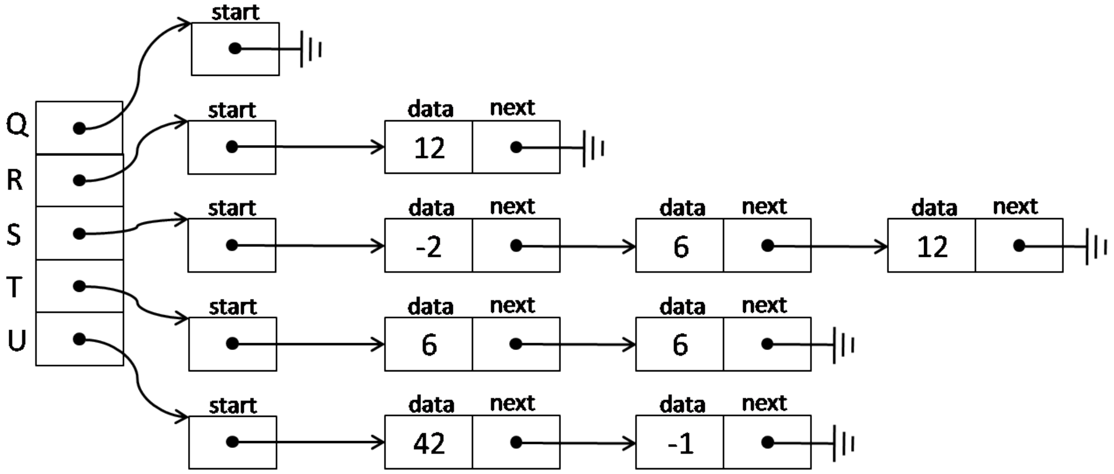
Grading: For full credit, figure out the problems in at least the first three broken implementations of sortedlist. For extra credit, figure out the problems in all the broken implementations.

Introduction

Hyrum's cloud-based motorcycle repair company *C0 on Wheels* is in trouble! In order to prepare for large number of clients and their motorcycles repairs, he implemented a new data structure to keep track of id numbers in sorted order. Unfortunately, all of his implementations seem to be having weird problems! He's hired YOU to figure out the problems with his code. The future of this SaaS (Scooter-repair as a Service) company is in your hands!

Sorted linked lists

Hyrum's data structure involves sorted linked lists of integers without duplicates. Another thing that's different from the linked lists that you've seen in lecture and homework is that there is no "dummy node" at the end of the list. The end of the linked list is reached when the `next` pointer on a node is `NULL`. Here's an example of Hyrum's data structure:



In the illustration above, `Q` is a sorted linked list containing no numbers, `R` contains just 12, and `S` contains −2, 6, and 12. Neither `T` nor `U` is a valid sorted linked list (for them, `is_sortedlist(T)` and `is_sortedlist(U)` will both return `false`).

The handout file `listlib.c0` contains declarations for the types `list` (identical to what we saw in class) and `sortedlist` (a struct pointing to a `list` as in the previous illustration). It also contains the following specification functions and helper functions, which may be useful while testing.

```
bool is_segment(list* start, list* end);
bool no_circularity(sortedlist* L);
bool is_sortedlist(sortedlist* L);
sortedlist* array_to_linkedlist(int[] A, int n)
  /*@requires 0 <= n && n <= \length(A) @*/ ;
int list_length(sortedlist* L)
  /*@requires L != NULL && no_circularity(L) @*/ ;
int[] linkedlist_to_array(sortedlist* L)
  /*@requires L != NULL && no_circularity(L) @*/
  /*@ensures list_length(L) == \length(\result) @*/ ;
bool arr_eq(int[] A, int[] B, int n)
  /*@requires 0 <= n && n <= \length(A) && n <= \length(B) @*/ ;
```

The function `array_to_linkedlist` **naively** constructs a linked list (not a sorted list) from an array. Thus, if we have an array `A` equal to `[-2, 6, 12]`, then `array_to_linkedlist(A, 3)` would create the linked list `S` above.

The handout directory `lab07` also contains Hyrum's five bad implementations of `sortedlist`, named `sortedlist-bad1.c0`, `sortedlist-bad2.c0`, etc. Each defines the functions `is_in`, `insert` and `delete`.

Your job for this lab will be to write exhaustive test cases for the functions `is_in`, `insert` and `delete` to catch the bugs in the broken implementations. Write your tests in the file `sortedlist-test.c0` in the directory `lab07`.

You can compile and run your code with these commands (one for each bad file):

```
% make 1
% make 2
% make 3
% make 4
% make 5
```

(`make` is a program that can help you compile code. You can Google it to learn more if you want.)

Your code should indicate a problem for each of the bad implementations. Figure out the exact line that causes that bug, and place a comment explaining the bug next to the line where it occurs. Do this for every bug you find in the code. Submit the code with your comments in Autolab for credit.

1.5pt **(2.a)** Find the bugs in the first broken implementation (`bad1`).

3pt **(2.b)** Additionally, find the bugs in the next two broken implementations (`bad2` and `bad3`).

4pt **(2.c)** Finally, find the bugs in the remaining two broken implementations (`bad4` and `bad5`).

Some hints:

- **To get the most out of this lab, don't spend a long time reading the bad implementations! Some of the bugs are quite subtle, and what we want to teach you is to write good tests.**

- Be thorough with your edge cases! Make sure the linked list behaves exactly as specified.
- Some implementations cause `NULL` pointer dereferences. Others cause contract failures. Others yet cause contract exploits. Make sure your tests can catch each of these types of bugs.
- Some bugs "cancel" each other out and make a list appear to work correctly and not fail any contracts. The later versions of `sortedlist` may have multiple errors!