CS251 Warmup Project: Linked lists / recursion

Due: Monday Sept 21 1:59PM

In this project you will complete a set of functions operating on sets of integers. The sets are represented as sorted sequences stored in singly linked lists.

You are given files `sset.h` and `sset.c`. Some of the functions listed in the .h file (the interface) are already completed in the .c file; the others have "stubs" which you are to complete.

Rules of the game / details:

Remember these are *sets* -- i.e., no duplicates.

You may not alter the .h file -- it simply specifies the interface.

This data structure is what we call "immutable": once a sorted set is created, it cannot be altered (things inserted; removed). *Then how can you do anything you may ask?* By having operations on sets produce *new* sets rather than modifying existing sets. For example, taking the union of two sets produces (returns) a new set.

Information hiding: notice that in the .h file the type `SSET` is aliased to `struct sset_struct`, but the actual definition of `sset_struct` is hidden in the .c file. By doing this we can hide the implementation of the actual data structures from client programs. The client programs can only operate on things of type `SSET *`. This gives us an Abstract Data Type (ADT) where the client only knows they name of the type and the operations that can be performed (by function calls). It has no direct access to the "guts" of the data structure.

Notice (in the `.c` file) that the struct is really a singly-linked linked list node. By convention, a **`NULL`** pointer is used to represent the empty set. ***However recall that the client has no knowledge of this convention!*** For example, a client program should not compare an **`SSET *`** with **`NULL`** to determine if it represents the empty set.

Recursion: The list representation of sorted sets is inherently recursive since the "tail" of a list is itself a list (perhaps NULL). This can be exploited to create very compact and elegant recursive code operating on sets.

Getting started

Download the source files `sset.h, sset.c` and `toy.c.` Also (***especially if you have no experience with multi-file compilation***), download and read the `readme` file.

Compile the files (again see the **`readme`** file) just to make sure everything is ok to start with.

Read through `sset.c`; you will see a number of function "stubs" labeled `TODO`. You are to complete these functions for the project. Notice that some are also given certain requirements that your implementation must follow.

The Functions

The .h file specifies the interface to the Sorted Set Abstract Data Type -- i.e., the operations that a client program can perform on sorted sets. The functions that you are to implement are:

`from_sorted_array` `(helper function called by sset_from_array)`
`sset_free`
`sset_contains`
`sset_intersection`
`sset_diff`
`sset_toarray`
`sset_cmp`
`sset_sort_sets` `(uses qsort to sort an array of sets according to the ordering criteria of sset_cmp)`

How to Proceed

We recommend that you develop your code using the following algorithm:

1. Select a function to implement.
2. Devise and implement a solution.
3. Devise test cases for the function and incorporate them into a driver program.
    a. Think about "corner cases"
    b. Make your test cases as small as possible while still being able to detect the type of bug they are designed to detect.
    c. Never delete test cases -- even after they have been "passed". Your test suite should always be growing.

    d. Exception to (c): if it turns out you have a bug in your test case! Then you fix it before moving on.
4. Evaluate your solution vs. the test cases you have created.
5. Debug your solution until it passes all of your test cases.
6. Passing all of your tests is a ***necessary condition*** for the correctness of your solution. Ask yourself *"am I confident that passing all of my tests tests is also a **sufficient condition** for correctness??"*
    a. If your answer is "YES" goto 1.
    b. If your answer is "NO" devise additional test cases and add them to your test suite and goto 4.

To repeat: think carefully about your test cases -- you are encouraged to share ideas with classmates on interesting test cases.

Additional Guidelines:

You may add helper functions to `sset.c` if you like, but there shouldn't be much of a need for many.

To repeat: You may ***not*** modify `sset.h` since it is the pre-specified interface that everyone must obey. Note that we will be writing our own driver programs to exercise/test your implementation so we must be able to link to all of your implementations in the same way.

What to Turn in

You will submit the following through blackboard:

- your `sset.c` file through blackboard.
- your tester program
- a readme file if there is anything unusual you'd like us to know