15-122: Principles of Imperative Computation

Lab 9: This one's a treet

Shyam Raghavan

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/lab-dict .
% cd lab-dict
```

You should add your code to the existing file `gen-dict.c1` and a new file `lcs-fast.c1` in the directory `lab-dict`.

Grading: Finish at least task (3.a) for 2 points, and additionally finish (4.a) for 3 points. Use the Makefile in the directory for testing your code. Once your code passes the tests, show it to a TA. Make sure you have reasonable contracts!

Computers are Hard

The Longest Common Subsequence problem is a famous $\mathcal{NP}$-hard problem in Computer Science theory that you'll be solving in polynomial time today (not really – we'll be solving a slightly easier problem using a technique called Dynamic Programming that you can learn much more about in 15-210 and 15-451)! We'll be using generic dictionaries implemented using binary search trees to memoize our solution and make it faster.

From Wikipedia, "The longest common subsequence (LCS) problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences). It differs from problems of finding common substrings: unlike substrings, subsequences are not required to occupy consecutive positions within the original sequences. The longest common subsequence problem is a classic computer science problem, the basis of data comparison programs such as the diff utility, and has applications in bioinformatics for comparing genome sequences. It is also widely used by revision control systems such as Git for reconciling multiple changes made to a revision-controlled collection of files."

Reviewing the BST implementation

Recall that we implemented a binary search tree in lecture as nodes, structs containing data and two (possibly `NULL`) pointers `left` and `right`.

```
1  typedef void* elem;
2  typedef int compare_fn(elem x, elem y)
3     /*@requires x != NULL && y != NULL; @*/;
4
5  typedef struct tree_node tree;
6  struct tree_node {
7     elem data;
8     tree* left;
9     tree* right;
10 };
11
12 typedef struct bst_header bst;
13 struct bst_header {
14    tree* root;
15    compare_fn* compare; // Function pointer must be non-NULL
16 };
```

Functions that the client will call are usually implemented as a function that takes a `bst*`. This function then calls an internal helper function that actually does the work, node by node, hiding the internal representation of the tree. These helper functions are often implemented recursively due to their naturally recursive nature.

A Generic Dictionary Implementation

(3.a) Update `gen-dict.c1` to implement a generic dictionary, adding in the necessary fields to the struct and altering each of the functions as needed.

Testing: `% make test-gen-dict`

Using a Generic Dictionary to Memoize the LCS Problem

Now that we have a working generic dictionary implementation, let's look back at the longest common subsequence problem.

We've got one implementation of an algorithm that solves the problem in `lcs-slow.c1`. To see how long this algorithm takes on two strings of 120 characters each, run `make time-lcs-slow` on the command line. Clearly, this takes a long time to run!

Now, we'll make the implementation faster by memoizing it using our dictionary implementation.

(4.a) In the `lab-dict` directory, create a new file called `lcs-fast.c1` and copy the code from `lcs-slow.c1` into it. Then, alter the code so that calls to `LCS` are memoized using the interface of the dictionaries you wrote earlier. When you're finished, note how much faster it runs than the slow version!

Testing: `% make time-lcs-fast`