15-122: Principles of Imperative Computation

Lab Week 9 — Tom Cortina, Rob Simmons

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. Feel free to talk with your neighbors about the problems!

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-bst .
% cd lab-bst
```

You should add your code to the existing file `bst.c1` in the directory `lab-bst`.

Grading: Finish at least task (2.a) for partial credit, and additionally finish (2.b) for full credit. The files `test-count.c1`, `test-countifless.c1`, and `test-height.c1` should be used for testing. Once your code passes the tests, show it to a TA. Make sure you have reasonable contracts!

Reviewing the BST implementation

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

```
typedef void* elem;
typedef int compare_fn(elem x, elem y)
   /*@requires x != NULL && y != NULL; @*/;

typedef struct tree_node tree;
struct tree_node {
   elem data;
   tree* left;
   tree* right;
};

typedef struct bst_header bst;
struct bst_header {
   tree* root;
   compare_fn* compare; // Function pointer must be non-NULL
};
```

Functions that the client will call are usually implemented as a function that takes a `bst*` (or, equivalently, a `bst_t`). 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.

Counting nodes in a binary search tree

(2.a) Complete the `bst_count` client function that has one parameter: the binary search tree `B` (of type `bst*`). The function returns the total number of elements currently stored in the binary search tree. Do not modify the data structure. Instead, traverse through the tree recursively, counting the nodes one by one, until you have visited all of the nodes.

Hint: In general, the number of nodes in a binary search tree is equal to 1 + the number of nodes in its left subtree + the number of nodes in its right subtree. (What is the base case?)

Testing: `cc0 -d -x bst.c1 test-count.c1`

In the previous exercise, we didn't care what elements were stored in the binary search tree; we just counted how many there were, which meant we didn't have to worry about the pointer to the client's `compare` function that is stored in the `bst_header`. For the next exercise, we will need to use the client-provided comparison function. For guidance, look at how the existing `bst_lookup` and `tree_lookup` functions do this.

```
elem tree_lookup(tree* T, elem e, compare_fn* compare)
//@requires compare != NULL && is_tree(T, compare);
//@ensures \result == NULL || (*compare)(\result, e) == 0;
{
   if (T == NULL) return NULL;
   int r = (*compare)(e, T->data);
   if (r == 0) {
      return T->data;
   } else if (r < 0) {
      return tree_lookup(T->left, e, compare);
   } else {
      //@assert r > 0;
      return tree_lookup(T->right, e, compare);
   }
}

elem bst_lookup(bst* B, elem e)
//@requires is_bst(B) && e != NULL;
//@ensures \result == NULL || (*B->compare)(\result, e) == 0;
{
   return tree_lookup(B->root, e, B->compare);
}
```

(2.b) Write a function `bst_countifless` that has two parameters: the binary search tree `B` (of type `bst*`) and an element `x` (of type `elem`).

This function returns the total number of elements currently stored in the binary search tree that are *strictly less than* the given element, as given by the client-provided comparison function.

Testing: `cc0 -d -x bst.c1 test-countifless.c1`

Height of a binary search tree

(3.a) Again without modifying the data structure (do *not* add a `height` field), complete the `bst_height` function that has one parameter `B` (of type `bst*`).

Remember that the height of a binary search tree is 0 if the tree is empty. Otherwise it is 1 + the height of its largest-height subtree.

Testing: `cc0 -d -x bst.c1 test-height.c1`

Comparing tree contents (Challenge problem)

(4.a) Write a function `bool bst_equal(bst* B1, bst* B2)` that checks if two trees have the same keys, even though they might have different structure.