15-122: Principles of Imperative Computation — Spring 2018

Lab 10: This one's a treet — 13 March

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-traversal .
% cd lab-traversal
```

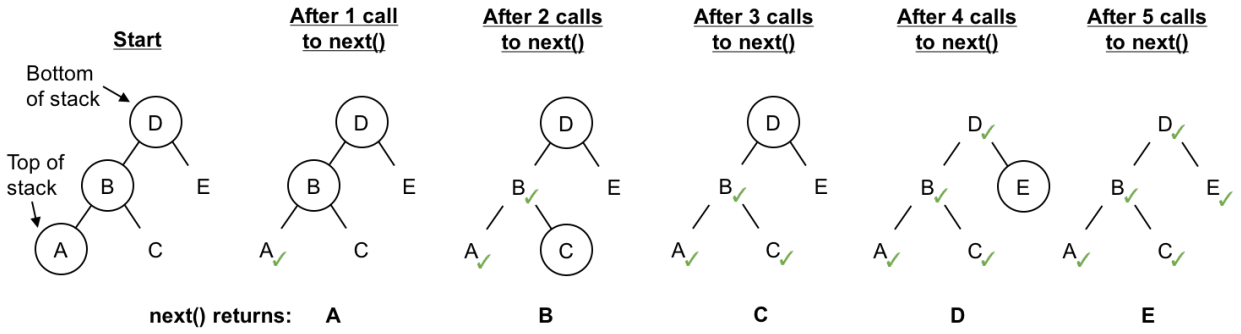
You should add your code to the existing file `bst.c1`.

Grading: Finish through task (3.b) for 2 points, and additionally finish (4.a) for 3 points. Make sure you have reasonable contracts!

In-order traversal

One of the most important properties of a binary search tree is that it maintains its elements in sorted order. The tree structure makes it easy to find, add, and remove elements in their correct position, but we haven't yet seen how to examine each element from smallest to largest. This is called an *in-order traversal*.

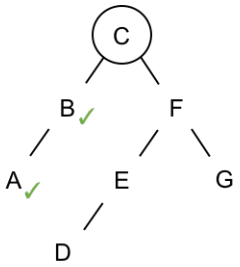
There are a few different ways to implement in-order traversal for a binary search tree. In this lab, we'll be using a stack to keep track of the nodes we still need to examine during traversal. Specifically, **whenever we follow the left child of a node, we push the node onto the stack** so we can come back to it later and visit its right subtree. Here's an example showing each step of a traversal (visited nodes have a green check mark next to them and the nodes on the stack are circled):



Notice how at each step, the next element we need to examine is at the top of the stack. Also notice that the `next()` function returns each of the values in sorted order.

(1.a) Suppose a traversal is in the state shown to the right of this text (with only node C in the traversal stack). What will the stack contain after the traversal is advanced by one? By two? Which values will be returned?

1pt



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`. This implementation is slightly different from what was shown in lecture — we're using **`void`**`*` as the `elem` type, and we're treating the entire element as a key.

```
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; // Non-NULL
};
```

Implementing the traversal

There are two parts to the in-order traversal implementation. First, we need a function that gives us the starting traversal stack (which represents the first element we need to look at). Once we have a traversal stack, we need a way to move ahead in the traversal to look at the next element.

(3.a) In file `bst.c1`, implement the function `bst_traverse_start`. This function returns the initial traversal stack that we'll use to begin our traversal (as in the first step of the diagram above). The stack should contain all the nodes on the path from the root to the minimum element, with the minimum element at the top of the stack.

(3.b) In file `bst.c1`, implement the function `bst_traverse_next`. Each time this function is called, the next smallest element in the tree is returned. Given a traversal stack, this function should do three things:

(a) Retrieve the data at the current node (which is at the top of the traversal stack)

(b) Modify the stack so that it represents the next node in the in-order traversal of the tree

(c) Return the retrieved data

Also, implement the one-line function `bst_traverse_finished`, which returns whether or not the given traversal stack has been advanced past the last element in the tree. This should be used in a precondition of `bst_traverse_next`.

2pt

You can test your code using: `cc0 -d -x lib/stack.c1 bst.c1 test-traverse.c1`.

Comparing tree contents

Since there are multiple valid binary search trees that contain the same elements, it is not possible to check if two binary search trees contain the same elements by just comparing their structure. In-order traversal can solve this problem.

(4.a) In file `bst.c1`, implement the function `bst_equal`, which returns whether or not two binary search trees contain the same elements. You may assume that the two binary search trees use the same comparison function.

3pt

You can test your code using: `cc0 -d -x lib/stack.c1 bst.c1 test-equal.c1`.