CS1XC3 - Computer Science Practice and Experience: Development Basics

Assignment #7 – Binary Search Trees

Weight

7% of total course grade

Due date

Sunday March 21st at 11:59pm

Primary Learning Objectives

- Write a C program that works with binary search trees (BSTs).

Requirements

As starter code for this assignment, use the C code contained in as7_starter_code.zip file found on Avenue to Learn accompanying this assignment.

This starter code contains the same binary search tree type used in lectures and labs, and the same set of functions for working with binary search trees that we created together in lecture and lab. The starter code also contains the following function declarations:

```
// HINT: This function was used as a helper function for array_of_sorted_keys
// in the instructor's solution:
// void build_array(bstNode *node, int *array, int *current_index);
int *array_of_sorted_keys(bstNode *node);

// HINT: This function was used as a helper function for
// balanced_tree_from_sorted_array in the instructor's solution:
// bstNode *construct_tree(int *array, int min, int max);
```

*bstNode *balanced_tree_from_sorted_array(int *sorted_array, int length);*

*bstNode *balanced_tree_copy(bstNode *tree);*

At a minimum you will need to implement the **array_of_sorted_keys**, **balanced_tree_from_sorted_array**, and **balanced_tree_copy** functions. Two function declarations have been commented out in the starter code, but they were used by the instructor to create their solution to the problems, and so you can implement these as well if you find them helpful (I would recommend implementing them).

The **array_of_sorted_keys** function must accept as an argument the head node of a BST. It needs to return a pointer to a dynamically allocated array that contains the keys of the BST sorted in ascending order. In order to do this, you'll need to dynamically allocate an array large enough to hold all of the keys. The **num_nodes** function can help you with this. In the week 9 lab you'll learn that you can print out the keys of a BST in an ascending sorted order if you do an in-order traversal of the tree. In this case, instead of printing the nodes, you'll need to store them in the dynamically allocated array that you've created.

In the instructor's solution, they used a helper function **void build_array(bstNode *node, int *array, int *current_index)** which performed an in-order traversal of the BST, storing the results into array, and using current_index to keep track of the index to place the next element visited into the array. You don't have to solve the problem exactly this way, but this way is recommended, and whatever you do, do not use a global variable to manage this.

The **balanced_tree_from_sorted_array** function needs to accept as arguments an array of ints stored in ascending sorted order, and the length of that array. It needs to return a pointer to a newly created ***balanced*** binary search tree that contains as keys all of the elements in the array (i.e. you are converting the array contents to a balanced binary search tree). A balanced binary search tree has the minimum height necessary to contains all of the keys. If you wanted to create a balanced tree from a sorted array, then the best node for the root node would be the middle element of the array, given that half the array elements will be less than this element and half the array elements will be greater than this element. If the array contains an even number of elements either of the two 'middle' elements will work. If you recursively applied this logic to each remaining left and right subsection of the array to build the left and right subtrees of this node, you could build a balanced binary search tree. So a function could work like this:

1. Is there still a middle to be found? If not, return NULL.
2. Get the middle of the array, create a new node with this value as the key.
3. Take the left half of the array, call the function recursively and set the left child of the node to the result.

4. Take the right half of the array, call the function recursively and set the right child of the node to the result.
5. Return the node.

In order to do this, the instructor's solution used a helper function **bstNode *construct_tree(int *array, int min, int max)** where min is the beginning index of the segment of the array being converted, and the max is the ending index of the segment of the array being converted. By recursively calling **construct_tree** with min and max values that reflect the portion of the array being converted for a given subtree, we can follow the above algorithm. We detect that there is no longer a "middle" that can be found when the function is called with a min > max. You do not need to solve the problem exactly this way, but this is the recommended approach.

The **balanced_tree_copy** function should accept as input a binary search tree that is potentially unbalanced. It should return a copy to a new balanced binary search tree on the heap which contains the same keys. In order to do this, you can call the **array_of_sorted_keys** function to obtain an array containing the keys of the binary search tree in ascending sorted order. Then you can call **balanced_tree_from_sorted_array** to obtain a balanced binary search tree from these results. Don't forget to free the array you created as there is no need for it after the function is done!

Test Code

The main function contains test code to test the 3 functions. You may want to "comment out" this code by surrounding it with /* … */ multiline comments so that you can work on your functions, as the code assumes that all of the functions have been implemented. You can use this code to help verify that your functions are working correctly.

The expected test code results are provided after the mark breakdown, and as a plaintext file on Avenue to Learn accompanying the assignment.

Submission

Save your solution in a file named **bst.c** and put it in a zip file named **a7.zip** and submit it to the Assignment #7 dropbox on Avenue to Learn.

Marking scheme

Component	Description	Marks
array_of_sorted_keys	Are the keys of the BST stored into an array that is returned? Are they sorted in ascending order? Is it don't without a	40

	global variable? Is it done with an in-order traversal of the list? Is it done reasonably efficiently and in a logical way?	
balanced_tree_from_sorted_array	Is a balanced tree created from the array? Does it conform to a BST? Is it done reasonably efficiently and in a logical way?	45
balanced_tree_copy	Is a balanced tree returned that contains all the same keys as the unbalanced tree? Are the two functions above used to do so?	15
Total:		100

Test Code Results

Tree before converted into a sorted array:

```
8
  3
    1
    6
      4
      7
  10
    14
      13
```

Array: 1 3 4 6 7 8 10 13 14

Tree constructed from the array:

```
7
```

```
  3
    1
    4
      6
  10
    8
    13
      14
```

Print out of an unbalanced tree:

```
0
  1
    2
      3
        4
          5
            6
              7
                8
                  9
```

Print out of the now balanced tree:

```
4
  1
    0
    2
      3
```

7

5

6

8

9