15-122: Principles of Imperative Computation, Spring 2020

Written Homework 9

Due on Gradescope: Tuesday 31st March, 2020 by 9pm

Name: ____

Andrew ID: ____

Section: ____

This written homework covers binary search trees and AVL trees.

Preparing your Submission You can prepare your submission with any PDF editor that you like. Here are a few that prior-semester students recommended:

- ***PDFescape*** or ***DocHub***, two web-based PDF editors that work from anywhere.
- ***Preview***, the Mac's PDF viewer.
- ***Acrobat Pro***, installed on all non-CS cluster machines, works on many platforms.
- ***iAnnotate*** works on any iOS and Android mobile device.

There are many more — use whatever works best for you. If you'd rather not edit a PDF, you can always print this homework, write your answers *neatly* by hand, and scan it into a PDF file — *we don't recommend this option, though*.

Submitting your Work Once you are done, submit this assignment on Gradescope. ***Always check it was correctly uploaded.*** You have unlimited submissions.

Question:	1	2	Total
Points:	7.5	4.5	12
Score:			

1. **Binary Search Trees**

1pt **1.1** Draw the final binary search tree that results from inserting the following keys in the order given. Make sure all branches in your tree are drawn *clearly* so we can distinguish left branches from right branches.

```
93, 86, 71, 115, 88, 99, 94, 77, 95, 109
```

2pts

1.2 How many different binary search trees can be constructed using the following five keys if they can be inserted in any order?

`42, 11, 59, -7, 83`

Show how your answer is derived. We've begun the derivation below; we've used $t(n)$ to stand for the number of binary search trees with n nodes.

Think recursively: How many trees with 0 nodes can possibly exist? How many different trees with 1 node can possibly exist? 2 nodes? 3 nodes? 4 nodes? Think about how to build up your answer from answers to simpler questions. (It might help to come back to this question after doing the last question on AVL tree height.)

n	$t(n)$
0	$t(0) = 1$
1	$t(1) = 1$
2	$t(2) = t(0) \times t(1) + t(1) \times t(0) = 2$
3	$t(3) =$
4	$t(4) =$
5	$t(5) =$

What is the general formula for $t(n)$ when $n > 0$?

$t(n) =$

For the next few questions, we consider the implementation of dictionaries as binary search trees in the lecture notes. In particular, recall the following declarations:

```
// typedef _______ key;
// typedef _______* entry;
key entry_key(entry e) /*@requires e != NULL; @*/ ;

bool is_bst(tree* T) {
  return is_tree(T) && is_ordered(T, NULL, NULL);
}
```

```
struct dict_header {
  tree* root;
};
typedef struct dict_header dict;
```

```
bool is_dict(dict* D) {
  return D != NULL
      && is_bst(D->root);
}
```

Like in class, the client defines two functions: `entry_key(e)` that extracts the key of entry `e`, and `key_compare(k1,k2)` that returns `-1` if key `k1` is "less than" key `k2`, `0` if `k1` is "equal to" `k2`, and `1` if `k1` is "greater than" `k2`.

1pt

1.3 Assume that the client also provides a function `entry_print(e)` that prints the given entry `e` in a readable format on one line. Complete the function `dict_print` that prints the entries in a dictionary on one line in order from smallest key to largest key. If the dictionary is empty, nothing is printed. You will need a **recursive** helper function `tree_print` to complete the task.

Think recursively: if you are at a non-empty node, what are the three things you need to print, and in what order? You should not need to examine the keys since the contract guarantees the argument is a BST.

```
void tree_print(tree* T)
//@requires is_bst(T);
{



}

void dict_print(dict* D)
//@requires is_dict(D);
{
  tree_print(________________________________);
  print("\n");
}
```

1.4 Consider extending the dictionary library implementation with the following function which deletes the entry with the given key k, if any.

```
void dict_delete(dict* D, key k)
//@requires is_dict(D);
//@ensures is_dict(D);
{
  D->root = bst_delete(D->root, k);
}
```

We will proceed in two steps.

1.5pts

a. Complete the code for the recursive helper function `largest_child` below which removes and returns the largest *child* rooted at a given tree node `T`. *(HINT: Finding the largest child of T actually doesn't require us to look at the keys. The largest child must be in one specific location.)*

```
entry largest_child(tree* T)
//@requires is_bst(T) && T != NULL && T->right != NULL;
{
  if (T->right->right == NULL) {

    entry e = ________________________________________;

    T->right = ________________________________________;
    return e;
  }

  return largest_child(________________________________);
}
```

2pts

b. Complete the code for the recursive helper function `bst_delete` on the next page which is used by the function `dict_delete` above. This function should return a pointer to the tree rooted at `T` once the entry is deleted (if it is in the tree). Note that this function uses the `largest_child` function you just completed.

```
tree* bst_delete(tree* T, key k)
//@requires is_bst(T);
//@ensures is_bst(\result);
{
  if (T == NULL)  return NULL;  // key is not in the tree

  int cmp = key_compare(k, entry_key(T->data));
  if (cmp > 0) {

    ___________________________ = bst_delete(T->right, k);
    return T;
  }
  else if (cmp < 0) {

    ___________________________ = bst_delete(T->left, k);
    return T;
  }
  else {  // key is in current tree node T
    if (T->right == NULL)

      return _____________________________________________;
    else if (T->left == NULL)

      return _____________________________________________;
    else {  // T has two children
      if (T->left->right == NULL) {
            // Replace T's data with the left child's data

         ________________________________________________;
            // Replace the left child with its left child

         ________________________________________________;
         return T;
      }
      else { // Search for the largest child in the left
            // subtree of T and replace the data in node
            // T with this data after removing the largest
            // child in the left subtree
        T->data = largest_child(T->left);
        return T;
      }
    }
  }
}
```

2. **AVL Trees**

2pts **2.1** Draw the AVL trees that result after successively inserting the following keys into an initially empty tree, in the order shown:

`E, J, N, L, X, K, T`

Show the tree after each insertion and subsequent re-balancing (if any) is completed: the tree after the first element, `E`, is inserted into an empty tree, then the tree after `J` is inserted into the first tree, and so on for a **total of seven trees**. Make it clear what order the trees are in.

Be sure to maintain and restore the BST invariants and the additional balance invariant required for an AVL tree after each insert.

2.2 Recall our definition for the height h of a tree:

> *The height of a tree is the maximum number of nodes on a path from the root to a leaf. So the empty tree has height 0, the tree with one node has height 1, and a balanced tree with three nodes has height 2.*

The minimum and maximum number of nodes m in a valid AVL tree is related to its height. The goal of this question is to quantify this relationship.

1pt

a. Let $m(h)$ be the minimum number of nodes in an AVL tree of height h. Fill in the table below relating h and $m(h)$:

h	$m(h)$
0	0
1	1
2	2
3	
4	
5	
6	

1pt

b. Guided by the table in part (i), give an expression for $m(h)$.
Here's a hint: recall that the nth Fibonacci number $F(n)$ is defined by:

$$
\begin{aligned}
F(0) &= 0 \\
F(1) &= 1 \\
F(n) &= F(n-1) + F(n-2), \qquad n > 1
\end{aligned}
$$

You may find it useful to use the Fibonacci function $F(n)$ in your answer. Your answer does not need to be a closed form expression; it could be a recursive definition like the one for $F(n)$.

0.5pts

c. Give a closed form expression (non-recursive) for $M(h)$, the *maximum* number of nodes in a valid AVL tree of height h.

$M(h) =$