15-122: Principles of Imperative Computation, Spring 2023

Written Homework 9

Due on Gradescope: Monday 20$^{\text{th}}$ March, 2023 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.
- ***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*.

Caution Recent versions of Preview on Mac are buggy: annotations get occasionally deleted for no reason. **Do not use Preview as a PDF editor.**

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:	9	6	15
Score:			

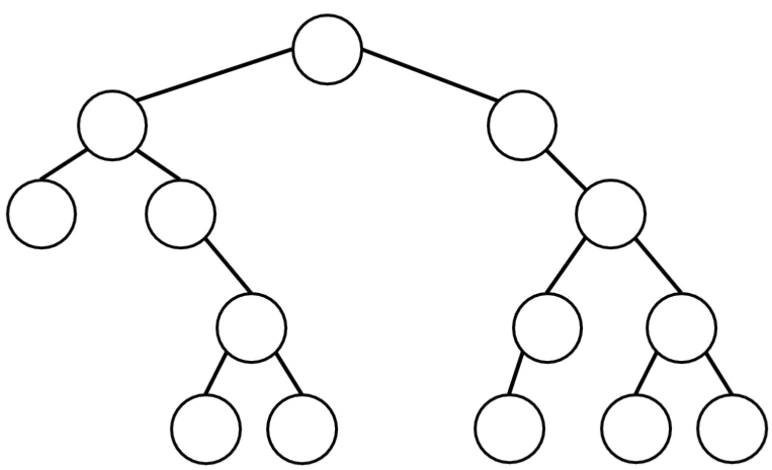
1. **Binary Search Trees**

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.

```
91, 85, 69, 110, 87, 98, 93, 76, 94, 108
```

1pt **1.2** Using the following keys, fill in the nodes of the tree below to obtain a valid BST.

```
0, 2, 3, 4, 5, 6, 8, 9, 11, 14, 15, 18, 20, 21
```



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; @*/ ;

typedef struct tree_node tree;
struct tree_node {
  entry data;        // != NULL
  tree* left;
  tree* right;
};
```

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

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

```
bool is_tree(tree* T) {
  if (T == NULL) return true;
  return T->data != NULL
      && is_tree(T->left)
      && is_tree(T->right);
}

bool is_ordered(tree* T, entry lo, entry hi)
//@requires is_tree(T);
{
  if (T == NULL) return true;

  key k = entry_key(T->data);
  return (lo == NULL || key_compare(entry_key(lo), k) < 0)
      && (hi == NULL || key_compare(k, entry_key(hi)) < 0)
      && is_ordered(T->left, lo, T->data)
      && is_ordered(T->right, T->data, hi);
}

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

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 a negative number if key `k1` is "less than" key `k2`, `0` if `k1` is "equal to" `k2`, and a positive number if `k1` is "greater than" `k2`.

1pt

1.3 Assume the client also provides a function `entry_print(e)` that prints entry `e` in a readable format on one line. Complete the function `dict_reverseprint` which prints the entries of the given dictionary on one line in order from largest key to smallest key. If the dictionary is empty, nothing is printed. You will need a **recursive** helper function `tree_reverseprint` 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_reverseprint(tree* T)
//@requires is_bst(T);
{



}

void dict_reverseprint(dict* D)
//@requires is_dict(D);
{
  tree_reverseprint(________________________________);
  printf("\n");
}
```

The function `bst_lookup` in the lecture notes is recursive, but it is also possible to implement it iteratively.

1pt

1.4 Fill in the missing code.

```
entry bst_lookup(tree* T, key k)
//@requires is_bst(T);
/*@ensures \result == NULL
        || key_compare(k, entry_key(\result)) == 0; @*/
{
  entry lo = NULL;        // to support the loop invariant
  entry hi = NULL;        // to support the loop invariant

  while ( _______________________________________________________

         _______________________________________________________)
  //@loop_invariant is_tree(T);
  //@loop_invariant is_ordered(T, lo, hi);
  {
    if ( ______________________________________________________ ) {
      hi = T->data;       // to support loop invariant
      T = T->left;
    }
    else {
      lo = T->data;       // to support loop invariant
      T = T->right;
    }
  }

  if (T == NULL) return NULL;
  return T->data;
}
```

The rest of this task verifies the safety and correctness of various parts of the above code using the methodology and format seen in this course. No or little credit will be given to answers that are unclear, verbose or unjustified.

1pt

1.5 Prove that the loop guard you wrote on lines 58–60 is safe — that pointer dereferences are safe and that the preconditions of any function you call are satisfied. You may assume that the loop invariant is correct. *(You may not need all lines.)*

______________________________ is safe because:

(a) ______________________ by ______________

(b) ______________________ by ______________

______________________________ is safe because:

(a) ______________________ by ______________

(b) ______________________ by ______________

(c) ______________________ by ______________

0.5pts

1.6 Prove that the loop invariant is true initially. You may assume it is safe.

To show: ______________________________ initially.

(a) ______________________ by ______________

(b) ______________________ by ______________

(c) ______________________ by ______________

(d) ______________________ by ______________

1pt

1.7 Prove that the loop invariant on line 62 is preserved by any iteration of the loop. Only show the case where the conditional on line 64 evaluates to `true`. Assume it is safe.

Assuming ______________, show that ______________

(a) ______________________ by ______________

(b) ______________________ by ______________

(c) ______________________ by ______________

(d) ______________________ by ______________

(e) ______________________ by ______________

(f) ______________________ by ______________

2.5pts

1.8 The function `bst_insert` in the lecture notes is recursive, but it is also possible to implement it iteratively. Fill in the missing code.

```
tree* bst_insert(tree* T, entry e)
//@requires is_bst(T) && e != NULL;
//@ensures is_bst(\result);
{
  key k = entry_key(e);
  tree* parent = NULL;
  tree* current = ________________________________________;
  while (current != NULL)
    /*@loop_invariant current == NULL || parent == NULL

                   || current == ________________________

                   || current == ________________________;@*/
    {
      parent = current;
      int cmp = key_compare(k, entry_key(current->data));
      if (cmp == 0) {
        current->data = e;
        return T;
      } else if (cmp < 0) {

        current = ________________________________________;
      } else { //@assert cmp > 0;

        current = ________________________________________;
      }
    }
  tree* R = alloc(tree);
  R->data = e;
  if (parent != NULL) {

    int cmp = ____________________________________________;
    if (cmp < 0)
      ____________________________________________________;
    else
      ____________________________________________________;
  }
  else {
    ______________________________________________________;
  }
  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**.

The BST ordering invariant is based on alphabetical order. Be sure to maintain and restore the BST invariants and the additional balance invariant required for an AVL tree after each insert.

AVL 1:	AVL 2:	AVL 3:
AVL 4:	AVL 5:	
AVL 6:	AVL 7:	

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. In particular, the empty tree has height 0.*

The minimum and maximum number of nodes n in a valid AVL tree is related to its height h. The goal of this question is to quantify this relationship and prove that $h \in O(\log n)$, i.e., that AVL trees are balanced.

1pt

a. Let $m(h)$ be the **minimum** number of nodes in an AVL tree of height h. Fill in the table to the right of this text relating h and $m(h)$.

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

0.5pts

b. Guided by this table, give an expression for $m(h)$. You may find it useful to use something similar to the Fibonacci function $F(k)$ in your answer. Recall its definition:

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

Your expression for $m(h)$ does not need to be a closed form expression; it could be a recursive definition like the one for $F(k)$.

1pt

c. It is possible to prove that $m(h) > g^h - 1$ for every $h > 0$, where g is the *golden ratio*, a very special real number whose value is $\frac{1+\sqrt{5}}{2} \simeq 1.618$. This fact allows you to show that $h \in O(\log n)$ for any AVL tree with n nodes and height h. *Note: the exact value of g is unimportant in this task.*

A. $m(h) > g^h - 1$	given
B. $n \geq m(h)$	by ________________
C. ______________________________	by ________________
D. ______________________________	by ________________
E. ______________________________	by ________________
F. ______________________________	by ________________
G. ______________________________	by ________________

Therefore, since $O(\log(n+1)) \subseteq O(\log n)$, we have that $h \in O(\log n)$.

2.3 Ben wants to streamline the code for AVL insertion seen in class. He rewrites it as the following wrapper function around the function `bst_insert` (insertion for *plain binary search trees*, not AVL trees). Assume `rotate_left` and `rotate_right` work as in the lecture notes, and that `height` returns the true height of the tree in constant time.

```
tree* new_avl_insert(tree* T, entry x)
// Macho programmers don't write contracts!  [see (*) below]
{
  T = bst_insert(T, x);
                      /* REBALANCE THE TREE */
  // Case: left subtree of T is too heavy compared to the right
  if (height(T->left) > height(T->right) + 1) {
    if (height(T->left->left) < height(T->left->right))
      T->left = rotate_left(T->left);
    T = rotate_right(T);
  }
  // Case: right subtree of T is too heavy compared to the left
  if (height(T->right) > height(T->left) + 1) {    // SYMMETRIC
    if (height(T->right->right) < height(T->right->left))
      T->right = rotate_right(T->right);
    T = rotate_left(T);
  }
  return T;
}
```

0.5pts

a. Draw the tree resulting from repeatedly calling `new_avl_insert` to insert the characters A, B, C, D, E, F in this order into an initially empty tree. Is this an AVL tree?

AVL tree?

☐ Yes

☐ No

1pt

b. An n-node tree is constructed using `new_avl_insert`. What is the complexity of calling `new_avl_insert` once more on it? In one sentence, justify why. (If you aren't sure, try inserting larger and larger entries in the last task.)

$O($____________________$)$ because ____________________________

____

(*) Disclaimer: *the real Ben would never say that. Write contracts!*