15-122: Principles of Imperative Computation, Spring 2024

Written Homework 9

Due on Gradescope: Monday 18^{th} March, 2024 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:

- **Kami**, **Adobe Acrobat Online**, or ***DocHub***, some 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*.

Please do not add, remove or reorder pages.

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:			

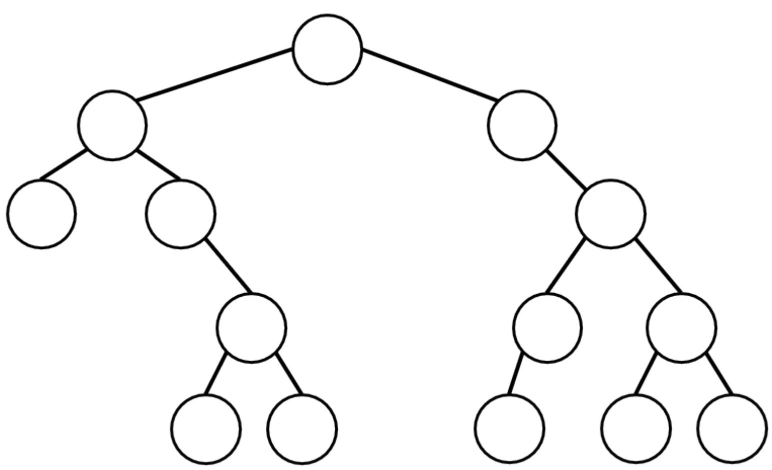
1. **Binary Search Trees**

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.

```
88, 79, 17, 122, 83, 105, 89, 42, 101, 112
```

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

```
0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91
```



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; @*/ ;
int key_compare(key k1, key k2);

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");
}
```

2.5pts

1.4 Write an implementation of a new dictionary library function, `dict_load`, that returns a measure of how long it will take to lookup or insert an entry. It does so by returning the height of the underlying binary search tree. The height of a binary search tree is defined as the maximum number of nodes as you follow a path from the root to a leaf. As a result, the height of an empty binary search tree is 0. Your function must include a **recursive** helper function `tree_height`.

HINT: In general, the height of a tree rooted at node T is one more than the height of its deepest subtree.

```
int tree_height(tree* T)

//@requires ________________________________;
//@ensures \result >= 0;
{













}

int dict_load(dict* D)

//@requires ________________________________;
//@ensures \result >= 0;
{
  return ________________________________;
}
```

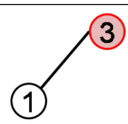
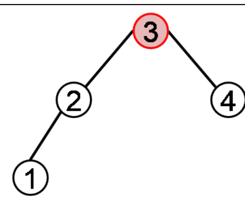
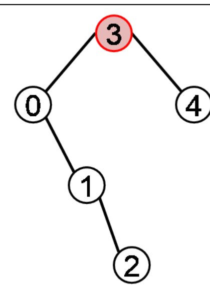
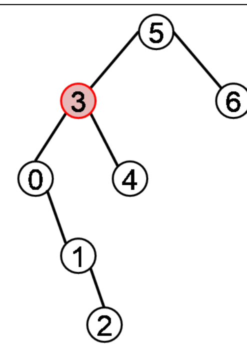
We want to extend the dictionary library implementation with the following function which deletes the entry with the given key k, if any.

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

The remaining tasks of this question implement `bst_delete`.

1pt

1.5 In each of the following BSTs, we want to delete node 3, and replace it with another node x from the tree such that the ordering invariant is maintained. Other than 3 and x, all other nodes should remain where they were. If we can replace 3 with multiple nodes, choose the one with the smaller value. In each case, draw the resulting BST.

	Case 1	Case 2	Case 3	Case 4
Original BST				
After deleting 3				

0.5pts

1.6 Complete the code for the recursive helper function `largest_descendant(T)` below which removes and returns the entry with the largest key in the right subtree of `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_descendant(tree* T)
//@requires is_bst(T) && T != NULL && T->right != NULL;
{
  if (T->right->right == NULL) {

    entry e = ____________________________________________;

    T->right = ___________________________________________;
    return e;
  }

  return largest_descendant(______________________________);
}
```

2pts

1.7 Complete the code for the recursive helper function `bst_delete(T,k)` which is used by the function `dict_delete` above. This function returns a pointer to the tree rooted at `T` once the entry with key `k` is deleted (if it is in the tree). The function `largest_descendant` you just completed as well as the cases in the first part of this question will come handy.

```
tree* bst_delete(tree* T, key k)
//@requires is_bst(T);
//@ensures is_bst(\result);
{
  if (T == NULL) return ______________________________;

/* Case 4 */ int cmp = key_compare(k, entry_key(T->data));
             if (cmp > 0) {
               ______________________________________;
               return T;
             }
             if (cmp < 0) {
               ______________________________________;
               return T;
             }

/* Case 1 */ //@assert cmp == 0;   // the root of T contains k

             if (T->right == NULL) return ________________;

             if (T->left == NULL)  return ________________;

/* Case 2 */             // 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;
             }
/* Case 3 */ 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:	

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 **2.2** The maximum number $M(h)$ and the minimum number $m(h)$ of nodes in a valid AVL tree of height h is given by the formulas below. For each, explain why this formula does indeed compute this number of nodes. Note that the formula for $m(h)$ is inductive.

Maximum number of nodes in an AVL tree of height h:

$$M(h) = 2^h - 1$$

because

Minimum number of nodes in an AVL tree of height h:

$$\begin{cases} m(0) &= 0 \\ m(1) &= 1 \\ m(h) &= 1 + m(h-1) + m(h-2) \quad \text{for } h > 1 \end{cases}$$

because

0.5pts

2.3 Consider an arbitrary AVL tree of height h containing n nodes. How are n and $m(h)$ related?

○ $n \leq m(h)$ ○ $n = m(h)$ ○ $n \geq m(h)$ ○ No relation

1pt

2.4 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 $h \in O(\log n)$.

Mihir 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 seen in class, 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

2.5 Draw the tree resulting from repeatedly calling `new_avl_insert` to insert the characters A, C, F, H, T, L in this order into an initially empty tree. Is this an AVL tree?

AVL tree?

☐ Yes

☐ No

1pt

2.6 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 Mihir would never say that. Write contracts!*