CSE/IT 122: Homework 8 - AVL Trees

Write a program that performs insertions, deletions, printing, and searching of a generic AVL tree. You will write three files – `avl.c`, `avl.h` which handles the AVL operations and `test.c` which tests the AVL tree.

You are given code (available on Canvas in `hw8.tar.gz`) that demonstrates insertion and inorder printing with a generic binary search tree (BST) and code that demonstrates insertion and inorder printing with an AVL tree, but the data structure is specific to the problem. You job is to write the following AVL tree functions that work with any type of data:

- avl_new /* create a new avl tree */
- avl_insert /* insert a node into an avl tree */
- avl_delete /* delete a node from avl tree */
- avl_free /* delete the entire avl tree */
- avl_find /* find a key and return its value */
- avl_inorder /* print the nodes in inorder traversal */
- avl_preorder /* print the nodes in preorder traversal */
- avl_postorder /*print the nodes in postorder traversal */

The above are not the only functions you need to implement: you have to implement the *user-defined* functions for printing, comparing, and freeing nodes as well as any helper functions (internal to `avl.c`) needed for the AVL functions.

NB you need to convert the `avl_node_t` structure in `avl.h` to a sentinel structure like the `BST_T` structure found in `bst.c`. When you do this, you will need to write helper functions. That is, a call to `avl_insert` will call an `insert` function that the end user doesn't see. You have to use helper functions as the end user only has a pointer to the sentinel node of `AVL_T` structure to work with. But what you want is the ability to access the nodes of the tree. That is an internal operation of your AVL tree and hidden from the user. The functions `bst_insert` and `bst_inorder`, in the file `bst.c`, give examples of helper functions.

Insertion into an AVL tree

The file `avl.c` contains a function `insert` which performs an insertion into an AVL tree. The issue is that it only works with an integer key and there is no information (value) associated with the key. It needs to be made generic. Rewrite the insertion so it works for any key-value pair using opaque, void, and function pointers.

The comparison function is user-supplied.

See `bst_insert` in the file `bst.c` for an example of a generic insert.

Delete nodes from an AVL trees

The file `avl.c` contains a function `delete_node` that is incomplete. As is the function performs a BST deletion. It needs to be modified so it balances the tree once a node is deleted. For the rotations, think about how the insertion code works.

You need to write the `find_max` function for the deletion.

The user needs to supply a free node function that frees the key and value of the node correctly. Use a function pointer.

No copying is allowed when you delete an interior node. Make sure when you delete an interior node you just move pointers rather than copy data into the new node and delete the leaf node.

Delete the AVL tree

Implement a means to delete the entire AVL tree and free all allocated memory. Use a post-order traversal.

Make sure you delete the tree and all memory is freed correctly before quitting your test program. The user needs to supply a free node function to free the key and value correctly (function pointer).

Test with Valgrind.

Traversals

Write generic in-, pre- and post-order traversals for the AVL tree. The file `bst.c` contains code for a generic in-order traversal. The traversals for an AVL tree are similar. The user supplies a user defined print node function (function pointer). Have the print node function print both the key and its value.

Finding a node

Write a function `avl_find` that finds a node based on the given key. Return the node's value. Finding is similar to the given BST insertion function *sans* the node creation.

The comparison function needed for find is user-supplied (function pointer).

Make sure to account for the case when the node is not found in the tree.

Test the AVL tree

The file `greek.txt` contains the key-value pairs you will use to test your AVL tree. The description of the Greek gods/mortals comes from Wikipedia. The file consists of the name of the Greek God/Mortal followed by a description. The name is the key and the description is the value.

Requirements

Perform the following operations in order and skip a couple of lines between output:

1. Insert the key-value pairs from the file into your AVL tree in the order of the file.
2. Print out the AVL tree using an inorder traversal. Make sure you print the key and its value.
3. Print out the AVL tree in preorder and postorder, but just print out the keys for this. You will have to write a different print function for this. Before you print the keys print out the type of traversal.
4. Find the node Dionysus and print the key-value pair. Dionysus should be found.
5. Find the node Jupiter and print an error message. This should fail as Jupiter is a Roman God.
6. Delete, in the order they are listed, the following nodes from the AVL tree:

   Hestia, Artemis, Hades, Pandora, Hephaestus, and Zeus
7. Print out the AVL tree (keys only) using an inorder traversal.
8. When you are satisfied with the output of you program, capture it with redirection. Name the file `avl.out`. For example

   ```
   $ ./test > avl.out
   ```
9. Make sure your code doesn't leak memory. Test with Valgrind.

Submission

Tar your source code files (`avl.c`, `avl.h`, `test.c` and `avl.out`) into a file named

`cse122_firstname_lastname_hw8.tar.gz`

and upload to Canvas before the due date.