CMSC 420: Coding Project 2
AVL Trees

1 Due Date and Time

Due to Gradescope by Sunday 5 March at 11:59pm. You can submit as many times as you wish before that.

2 Get Your Hands Dirty!

This document is intentionally brief and much of what is written here will be more clear once you start looking at the provided files and submitting.

3 Assignment

We have provided the incomplete file `avl.py` which you will need to complete. More specifically you will fill in the code details to manage height calculation, insertion, bulk deletion, search, dump, and range query for AVL trees. Unlike the binary search trees, these AVL trees contain both a key and a value (a string in this case). Please look at this file as soon as possible.

4 Details

The functions should do the following:

- **`def height(root: Node) -> int:`**
  Calculate the height of the tree rooted at `root`.
- **`def insert(root: Node, key: int, value: str) -> Node:`**
  Insert a key-value pair and rebalance the tree. The key is guaranteed to not exist.
- **`def delete(root: Node, keys: List[int]) -> Node:`**
  Perform a bulk deletion of all the nodes whose keys are in the list. They are all guaranteed to exist. Do this by first tagging all the corresponding nodes somehow, then perform an in-order traversal of the tree, ignoring the tagged nodes, to get an ordered list of key-value pairs which remain, then completely rebuild the tree using that list and `insert`.

- `def search(root: Node, search_key: int) -> str:`

  For the tree rooted at root, calculate the number of keys on the path from the root to the search key, including the search key, and the value associated with the search key. Return the json stringified list of the form `[number of keys, corresponding search value]`. If the search key is not in the tree return `[number of keys, None]`.

- `def rangequery(root: Node, x0: int, x1: int) -> List[str]:`

  Perform a range query for all keys in the tree between `x0` and `x1` inclusive. Return a list of *corresponding values*. Careful! Do this by creating a list and appending recursively as follows, starting at the root: If the key is in the interval, append its value and then recursively look at both children. If the key is greater than `x1` then recursively look at the left child and if the key is less than `x0` then recursively look at the right child. Return the list of corresponding values. Note that if you do this in a different way you may get an incorrectly ordered list.

5 Additional Functions

You will probably need some helper functions as in the first project and you will also need to write the functions to do the necessary rotations which balance the tree and an inorder traversal (which you can steal and tweak from the first project).

6 What to Submit

You should only submit your completed `avl.py` code to Gradescope for grading. We suggest that you begin by uploading it as-is (it will run!), before you make any changes, just to see how the autograder works and what the tests look like. Please submit this file as soon as possible.

7 Testing

This is tested via the construction and processing of tracefiles just like the first project.

Each non-final line in a tracefile is either `insert,key,value` or `delete,k1,k2,k3,...` (indeterminate number of keys). All together these lines result in an AVL tree.

The final line is one of the following: `dump`, `height`, `search,x` (where `x` is a key), or `rangequery,x0,x1` (where `x0` and `x1` give a range). This final line determines which of these operations is carried out.

You can see some examples by submitting the `avl.py` file as-is.

8 Local Testing

We have provided the testing file `test_avl.py` which you can use to test your code locally. Simply put the lines from a tracefile (either from the autograder or just make one up) into a file `whatever` and then run:

```
python3 test_avl.py -tf whatever
```