CMSC 420: Coding Project 1
Binary Search Trees

1 Due Date and Time

Due to Gradescope by Sunday 19 February 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 `bst.py` which you will need to complete. More specifically you will fill in the code details to manage insertion, deletion, search, dump, and four types of traverses for binary search trees. Please look at this file as soon as possible.

4 Coding Primer

In this assignment we are not creating and working with a class for an entire tree but just for a node, as shown here by this snippet of code from `bst.py`:

```
class Node():
    def  __init__(self,
                  key        = None,
                  leftchild  = None,
                  rightchild = None):
        self.key        = key
        self.leftchild  = leftchild
        self.rightchild = rightchild
```

A tree will simply consist of a collection of node instances with their children set correctly. We will reference a tree simply by referencing the instance of its root node.

Thus we can start building a tree via a command such as:

```
root = Node(key=100)
```

Then we can add a left child to this via:

```
root.leftchild = Node(key=50)
```

And we could delete this same child via:

```
root.leftchild = None
```

Note 4.1. Python does automatic garbage collection so when we delete the final reference to a variable, such as the left child above, the memory is freed.

5 What to Submit

You should only submit your completed `bst.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.

6 Testing

There are numerous tests which build from simple to more complicated. You'll see them described in detail when you do your first submission.

Each test randomly constructs a tracefile; this is a file containing a line-by-line set of instructions finishing with some sort of request which generates output. This tracefile is then processed using the functions in your code and your result is compared to the correct result. Each test is all-or-nothing for points earned.

Each non-final line in a tracefile specifies either an insert or delete. All together these lines result in a binary tree.

The final line is one of the following: `dump`, `preorder`, `inorder`, `postorder`, `bft`, or `search,x` (where `x` is a key). This final line determines which of these operations is carried out.

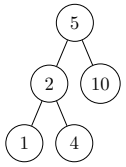
Example 6.1. For example consider the following tracefile:

```
insert,5
insert,10
insert,2
insert,3
insert,4
delete,3
insert,1
dump
```

The lines would be processed in the order given where the final line then dumps the tree to stringified JSON. The result would look like this:

```
{"k": 5, "l": {"k": 2, "l": {"k": 1, "l": null, "r": null},
"r": {"k": 4, "l": null, "r": null}}, "r": {"k": 10, "l": null, "r": null}}
```

This represents the tree:



Note 6.1. To see more examples, submit the `bst.py` file as-is.

Note 6.2. You do not need to write the code to process the tracefiles. The tracefiles are simply there to show you what is being done and to allow you to do some offline testing. All you need to do is fill in the code in `bst.py`.

7 Local Testing

We have provided the testing file `test_bst.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_bst.py -tf whatever
```