CMSC 420 Fall 2024
Coding Project
Binary Search Trees Version 2

1 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.

2 Assignment

We have provided the incomplete file `bst.py` which you will need to complete. Please look at this file as soon as possible.

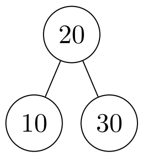
More specifically you will fill in the code details to manage insertion, deletion, and search for a modified version of a binary search tree. The binary search tree is organized by keys and each key has an associated value. The keys are guaranteed be distinct.

The variation is as follows. Here we just talk about keys for simplicity but the values go with the keys.

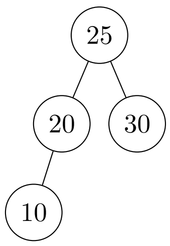
- Insert: We are modifying insert to try to keep newly inserted keys close to the root.

  Typically insert works by traveling all the way to a leaf and then adding the new key. In this variation as we travel down the tree if we come across a node (maybe even the root) whose current key can be safely replaced by the new key (meaning the new key is greater than every key in the left subtree, if one exists, and less than every key in the right subtree, if one exists) then we replace the node's old key by the new key and send the old key down the correct branch as directed by the new key and after that branch we proceed with insertion using the old key. Note that this new insertion process with the replaced key also abides by the same replacement rules.

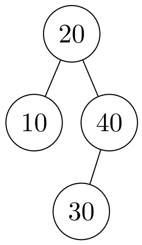
  For example if a tree looks like this:



If we insert 25 we start at the root and note that 25 can replace 20 (no conflict with the subtrees) so we replace the 20 by 25 and then proceed as if we were inserting 20. Since $20 < 25$ we go left. Now we are at 10 and 20 can replace this (no conflict with subtrees since it has no subtrees) so we replace the 10 by 20 and proceed as if we were inserting 10. Since we're at a leaf and $10 < 20$ we insert left:



On the other hand if we insert 40 in the original tree we start at the root and note that 40 cannot replace 20 (conflict with the subtrees) so since $40 > 20$ we go right as usual. Now we are at 30 and 40 can replace this (no conflict with subtrees since it has no subtrees) so we replace the 30 by 40 and proceed as if we were inserting 30. Since we're at a leaf and $30 < 40$ we insert left:



- Delete: We are modifying delete to try to keep the tree a bit more balanced.

  As usual if the node to be deleted has one child we simply splice.

  In the two-child case we need to find a replacement node check which subtree is larger (number of nodes). If the left subtree has more nodes then use the inorder predecessor as the replacement, otherwise use the inorder successor as the replacement. In this way we are essentially deleting from the subtree which has more nodes.

3 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        : int  = None,
                  value      : int  = None,
                  leftchild  : Node = None,
                  rightchild : Node = None):
        self.key        = key
        self.value      = value
        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,value=42)
```

Then we can add a left child to this via:

```
root.leftchild = Node(key=50,value=42)
```

And we could delete this same child via:

```
root.leftchild = None
```

Note 3.1. Python does automatic garbage collection so when we delete the final reference to a variable, such as the left child above, the memory is automatically freed.

4 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.

5 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 an insert or delete. All together these lines result in a binary tree.

The final line is either `dump` or `search,x` (where `x` is a key). This final line determines which of these operations is carried out.

Note 5.1. 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`.

6 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
```

Note that this will not tell you what the answers should be (because then we'd have to make the correct code available) but will allow you to debug and perform small tests.