# 420-LCW-05 Programming Techniques and Applications - Lab 6

March 16, 2022

This lab assignment, like others in the course, will not be formally graded. I do NOT expect you to finish all of the exercises, but do think about them and see if you can imagine how a solution might work. I guarantee that some elements of these labs will appear on future examinations!

## 1 Introduction

In this lab you will write some code to work with binary search trees (BST). I've provided a very basic implementation of a BST - it is up to you to extend it in a number of ways. In addition, you will run some tests on the BST class and see how it performs in typical situations.

Remember that a BST is a structure in which *nodes* contain a *key* and an optional *value* that is associated with the key. In addition, each *node* contains links to the left and/or right subtrees. The rule in a BST is that all keys on a node's left subtree must be less than the key of the node, and all keys on the node's right subtree must be greater than the key of the node. In general, assume that *no duplicate keys are permitted.*

## 2 Existing code

The code provided in `BST.py` implements a BST class with several methods provided:

- `__len__(self)` - special method that implements the `len()` function.
- `__bool__(self)` - special method that specifies how a BST is converted to `bool`.
- `_size(node)` - static method that computes the size of the subtree rooted at `node`.
- `put(self, key, value)` - method to add or update a key/value mapping.
- `_get(node, key)` - static method that finds the node associated with the key.
- `get(self, key)` - returns the value associated with the key, and raises `KeyError` if not found.
- `__contains__(self, key)` - returns `True` if key is present in the BST. Used to implement the `in` operator.
- `depth(self)` - returns the maximum depth of the BST.

## 3 Your tasks

### 3.1 Fixing `_get()`

The private `_get(self, key)` method is used by both the public `get(self, key)` and `__contains__(self, key)` methods. The implementation provided is correct but unnecessarily slow because it is recursive instead of iterative. Rewrite the method to use a single `while` loop instead of recursion.

## 3.2 Adding `minKey(self)` and `maxKey(self)`

Add methods `minKey()` and `maxKey()` that return the smallest (leftmost) and largest (rightmost) keys in the BST. You may write these either iteratively or recursively, as you prefer.

## 3.3 Inorder Traversal

Add a method called `traverse(self, func)` that traverses the tree using *in-order* traversal. Your code will be nearly the same as what was presented in the slides, except your function should expect two arguments, e.g. it should be called as `func(key, value)`. Your method should call the `func` on each key/value pair in *ascending key order*.

## 3.4 Adding support for `repr`

Your job is to write the method necessary to convert the *entire tree* into a useful string. As a model, imitate the format of a Python dictionary:

```
bst = BST()
bst.put('a', 0)
bst.put('d', 11)
bst.put('c', 0)
bst.put('b', 10)
print(bst)
{'a':0, 'b':10, 'c':0, 'd':11}
```

**Unlike** a Python dictionary, your method *must* list the keys in ascending order. In all other respects the format should mimic a Python dictionary exactly.

You *must* use your `traverse()` method from the previous part to implement this. The trick is to create a function that will build up the correct string during the tree traversal. Do **not** create an actual Python `dict` anywhere in your code. *Hint* consider creating a local helper function to build a `list` that you can assemble with `join()`.

## 3.5 Testing and performance

Open the file `BSTtest.py` and add code to test all of your new methods. Be sure that you do not just print results, but compare the results you get to the “correct” values and make sure they work (using `if` or `assert` statements).

I have provided some additional code to investigate an aspect of the performance of BSTs. It prints the minimum, maximum, and average *depth* for trees of a given size, for the sizes $N$ = 15, 31, 63, 127, and 255.

The process computes a list of 100 tree depths for each value of $N$. It prints out the minimum, maximum, and mean depths in a nicely-formatted table, similar to this:

```
N   MIN MAX  MEAN
 15   X  YY  Z.ZZ
 31   X  YY  Z.ZZ
 63   X  YY ZZ.ZZ
127  XX  YY ZZ.ZZ
255  XX  YY ZZ.ZZ
```

Note that each time you run the code, it might give slightly different values for this table. That's because it is randomizing the data, and produces a different sequence of BST objects each time it runs.

The performance of BSTs depends strongly on the depth of the tree. In the comments of your modified `BSTtest.py` file, write a few sentences that:

- Describe the best case and worst case behavior expected for a given value of $N$. Remember that this is related to the minimum and maximum depth of the BST for a given $N$.
- Comment generally on the observed relationships between $N$ and the observed minimum, maximum, and average tree depths.
- Discuss the implications of your observations for BST performance.

## 3.6 *Optional but easy*: support indexing

One nice thing about Python is that we can implement "special" methods in order to provide seamless integration between our classes and the language.

Implement the methods `__getitem__(self, key)` and `__setitem__(self, key, val)`. These allow you to use indexing with your BST class:

```
bst = BST()
bst['a'] = 1
bst['c'] = 2
bst['b'] = bst['a'] + bst['c']
print(bst)
{'a':1, 'b':3, 'c':2}
```

## 3.7 *Optional but tricky*: support comparison

Obviously it should be possible to provide an `__eq__` method to compare two BSTs. Two trees with the same *contents* should be equal. However, this is not trivial because the tree structure depends on the order in which keys are added. What techniques could you use to solve this? And can you define the other comparisons on BSTs? Don't worry too much about efficiency!

## 3.8 *Really optional*: deletion

Deletion in a BST is tricky. Implement a `delete(self, key)` method that deletes a key using Hibbard deletion.

## 4 What to hand in

Submit a ZIP file consisting of:

1. Your modified BST.py file.
2. Your modified BSTtest.py file, with comments answering the questions given above.