# CS 3430: SciComp with Py
# Assignment 9
# Estimating Probabilities of Linearity and Balance in Random Binary Search Trees

Vladimir Kulyukin
Department of Computer Science
Utah State University

November 4, 2017

## 1 Learning Objectives

1. Binary Search Trees
2. Conditional Probabilities
3. Probability Distributions
4. Plotting Data with Matplotlib
5. OOP

## 2 Introduction

This assignment starts our transition to machine learning (ML). You'll learn (or review if you've already studied it) one of the most useful data structures in CS - the binary search tree (BST). We'll discuss some theory behind BSTs, implement a few methods of the BST class and then use it to estimate the probability of a randomly generated BST being a list, i.e., being a *linear tree*, or being a *balanced* BST. If you are not familiar with these terms, don't worry - they're all defined below. The techniques of estimating this likelihood will serve as a gentle introduction to conditional probability, a very useful concept frequenty and productively used in ML. And while you're at it, you'll play with Matplotlib and get more exposure to OOP.

## 3 Some Definitions

A BST is a special kind of graph that consists of nodes and edges. The nodes store *keys.* For the purposes of this assignment, we'll assume that the keys are numbers. The term *binary* in BST means that in this graph each node has at most two child nodes (hence *binary*). The graph can also be empty, i.e., have no nodes and no edges. In this case, we say that the BST is empty. The term *tree* in BST means that there's a unique node, called the *root*, that doesn't have any parent, i.e., it's a child of no other node. If a node has no children, it's called a *leaf.*

The term *search* in BST means that a BST is a search tree, because it imposes an additional constraint on the values of the node keys: for each node in the tree it is true that the keys in all nodes to the left of it are strictly less than the node's key whereas the keys in all nodes to the right of it are strictly greater than the node's key. If the binary tree is a search tree and we're looking for a key in it, at any given node either the node's key is equal to the key we're looking for or it is to the left of it or to the right of it. Of course, if we're at a leaf node and its key is not equal to the key we're looking for, we know that the tree doesn't contain it.

```
10      10       10         10        10
       /        /\         /\          \
      5        5 20       5 20          20
                         /   /\          \
                        4   15 35         30
```

The trees above are all BSTs. The leftmost BST consists of one node, 10, which is both the root and the leaf. These trees are binary, because in each tree each node has at most two children. These trees are all BSTs, because for each node's key the keys in the nodes to the left of it are strictly less whereas the keys in the nodes to the right of it are strictly greater. For example, consider key 10 in the second tree from the left, key 5 is to the left of key 10 and key 20 is to the right of it and $5 < 10$ and $20 > 10$. Note that the search condition must hold true for *every* node in the tree. For example, the following tree is a binary tree but not a BST:

```
  10
  /\
 5  20
    / \
   30 50
```

Why? Because the key condition is not satisfied at the node with key 20: 30 is to the left of it, but $30 > 20$.

## 4 BST Nodes

As mentioned above, a BST consists of BST nodes. Each node contains a key. A blueprint of the BST node class is given in `BSTNode.py`. You can use this class as is without any modification. It has a constructor, a few getters and setters and implements the magic method `__str__()`. Here is a quick example of how you can use this class to construct BST node objects.

```
>>> bn1 = BSTNode(key=10)
>>> str(bn1)
'BSTNode(key=10, lc=NULL, rc=NULL)'
```

What we just did above is constructing a BST node `bn1` with a key of 10 and its left and right children being `None`. When we apply `str()` to it, it calls the magic method `__str__()` and produces a string that states that the key of `bn1` is 10, its left child (`lc`) is `NULL`, and its right child (`rc`) is `NULL` as well. Let's set the left child of `bn1` to a node with a key of 5 and the right child of `bn1` to a node with a key of 20:

```
>>> bn1.setLeftChild(BSTNode(key=5))
>>> bn1.setRightChild(BSTNode(key=20))
>>> str(bn1)
'BSTNode(key=10, lc=+, rc=+)'
```

If we evaluate `str(bn1)`, it returns a string that indicates that both the left and right children are defined, i.e., `lc=+` and `rc=+`. The `+` simply means that the corresponding child node is not `None`. If we want to access and print the left and right children of `bn1`, we can do it as follows:

```
>>> str(bn1.getLeftChild())
'BSTNode(key=5, lc=NULL, rc=NULL)'
>>> str(bn1.getRightChild())
'BSTNode(key=20, lc=NULL, rc=NULL)'
```

## 5 BST

The file `BSTree.py` has a blueprint of BST class. As the constructor `__init__()` of the `BSTree` class indicate, a `BSTree` object has two attributes: the root node (possibly empty, i.e., `None`) and the number of nodes in the tree:

```
def __init__(self, root=None):
  self.__root = root
  if root==None:
    self.__numNodes = 0
  else:
    self.__numNodes = 1
```

The method `insertKey(self, key)` inserts a key into a `BSTree` *only* when the key is not in the tree. Some definitions of BSTs allow for the presence of duplicates, i.e., nodes with the same key. However, for the sake of simplicity and clarity, in this assignment we'll consider BSTs with no duplicates. The method `insertKey(self, key)` returns `True` when the node is successfully inserted and `False` if the node's key is already in the BST. If the tree has no nodes, the key becomes the root's key. Otherwise, we start walking down the tree. First, we compare the key to the key of the current node. Intially, the current node is set to the root. If the current node's key is less, we set the current node to its left child and keep going down from there. If the key is greater, we set the current node to its right child and keep going down. Eventually, we find a leaf node, i.e., a node without any children and insert the key either to the left or right of that leaf node, depending on whether the key is less or greater than the leaf's key. Let's create a BST and insert three keys into it: 10, 5, and 20.

```
>>> bst = BSTree()
>>> bst.insertKey(10)
True
>>> bst.insertKey(10)
False
>>> bst.insertKey(5)
True
>>> bst.insertKey(20)
True
>>> str(bst.getRoot())
'BSTNode(key=10, lc=+, rc=+)'
>>> str(bst.getRoot().getLeftChild())
'BSTNode(key=5, lc=NULL, rc=NULL)'
>>> str(bst.getRoot().getRightChild())
'BSTNode(key=20, lc=NULL, rc=NULL)'
```

The method `displayInOrder(self)` displays the tree nodes in order, i.e., for each node, the nodes in the left subtree are displayed first, then the node itself, and then the nodes in the right subtree. Here's what `displayInOrder` prints out when called on the newly constructed BST:

```
>>> bst.displayInOrder()
NULL
BSTNode(key=5, lc=NULL, rc=NULL)
NULL
BSTNode(key=10, lc=+, rc=+)
NULL
BSTNode(key=20, lc=NULL, rc=NULL)
NULL
```

`NULLs` are displayed for empty nodes. This in-order display corresponds to the following BST, where `x` denotes the empty node, i.e., `NULL`.

```
                  10
                  /\
                 5  20
                /\  /\
               x x  x x
```

## 6 Height of a Binary Search Tree and Its Linearity

The height of a BST node is defined recursively as follows. The height of a `NULL` node (`None` in the Py jargon) is -1. The height of a BST node with a key but no children, i.e., a leaf node, is 0. Otherwise, the height of a BST node is 1 + max(height of the node's left child, height of the node's right child). The height of a BST is the height of its root. Let's get back to the BST examples discussed above.

```
    10     10      10        10        10
           /       /\        /\         \
          5       5 20      5 20        20
                           /  /\         \
                          4  15 35       30
```

Going left to right, the height of the 1st tree is 0, the height of the 2nd tree is 1, the height of the 3rd tree is also 1, the height of the 4th tree is 2, and the height of the 5th tree is 2 as well. A BST is linear, i.e., a list, when the number of nodes in it is exactly 1 greater than its height. For example, the 2nd and 5th trees are linear.

In `BSTree.py`, implement the method `isList(self)` that returns `True` when the BST is a list and `False` otherwise. You'll also need to implement the method `heightOf(self)` that returns the height of the BST. Here is a quick example where we construct the following BST that has 3 nodes and whose height is 2.

```
           10
          /
         5
          \
           9
```

```
>>> bst = BSTree()
>>> bst.insertKey(10)
True
>>> bst.isList()
True
>>> bst.insertKey(5)
True
>>> bst.isList()
True
>>> bst.insertKey(9)
True
>>> bst.isList()
True
>>> bst.getRoot().getLeftChild().getRightChild()
<BSTNode.BSTNode instance at 0x7feacd0ebb00>
>>> str(bst.getRoot().getLeftChild().getRightChild())
'BSTNode(key=9, lc=NULL, rc=NULL)'
>>> bst.getNumNodes()
3
>>> bst.heightOf()
2
```

## 7 Balance

A BST is *balanced* if for every node the heights of the left and right children differ by at most 1, which can be measured by the absolute value of the children's heights. In the five trees above, the first four trees are balanced. The $5^{th}$ tree is not, because the balance factor is broken at node 10. The height of the node's left child is -1 and the height of right child is 1. If we take the absolute value of the difference between the left child's height and the right child's height, i.e., $|-1-1| = 2$, we see that this tree is not balanced at node 10. In `BSTree.py`, implement the method `isBalanced(self)` that returns `True` if the BST is balanced and `False` otherwise.

## 8 Random Binary Search Trees

Now that we have `isList(self)` and `isBalanced(self)`, let's start experimenting with probabilities. Toward that end, implement a function `gen_rand_bst(num_nodes, a, b)` to generate random BSTs. The first argument specifies the number of nodes in a BST, the second and third arguments specify the range in which the random numbers are generated and inserted into the tree until the tree has the required number of nodes. The method returns a BST object. Let's generate a random BST with 5 nodes whose keys are random integers in $[0, 100]$ and check if it's a list.

```
>>> rbst = gen_rand_bst(5, 0, 100)
>>> rbst.isList()
False
>>> rbst.isBalanced()
BSTNode(key=70, lc=+, rc=+) not balancedFalse
>>> rbst.displayInOrder()
NULL
BSTNode(key=23, lc=NULL, rc=+)
NULL
BSTNode(key=33, lc=NULL, rc=+)
NULL
BSTNode(key=43, lc=NULL, rc=NULL)
NULL
BSTNode(key=70, lc=+, rc=+)
NULL
BSTNode(key=75, lc=NULL, rc=NULL)
NULL
```

The above in-order display displays the following BST:

```
    70
   /  \
  23   75
   \
    33
```

```
        \
        43
```

## 9 Estimating Probabilities of Linearity and Balance

Implement the function

```
estimate_list_prob_in_rand_bsts_with_num_nodes(num_trees, num_nodes, a, b)
```

The function returns the probability of a random BST being a list by generating `num_trees` of random BSTs, each of which has `num_nodes` of random numbers in `[a, b]`, counting the number of linear BSTs among the generated random BSTs and dividing that number by `num_trees`, i.e., it estimates the probability of a BST being linear given that it has the number of nodes specified by the second argument and each key is a random number in $[a, b]$. This is a *conditional probability* in that the probability of a BST being a list is *conditioned* on the tree having a specific number of nodes whose keys reside in a specific range. The function should return a tuple whose first element is the conditional probability and the second element is the actual list of generated binary search trees that are lists. Here is a test run.

```
>>> estimate_list_prob_in_rand_bsts_with_num_nodes(100, 5, 0, 1000)
(0.13, [<BSTree.BSTree instance at 0x7f09fde98518>, <BSTree.BSTree instance at 0x7f09fde98098>,
<BSTree.BSTree instance at 0x7f09fdf02b00>, <BSTree.BSTree instance at 0x7f09fde9e050>,
<BSTree.BSTree instance at 0x7f09fde9c8c0>, <BSTree.BSTree instance at 0x7f09fde9ce60>,
<BSTree.BSTree instance at 0x7f09fde944d0>, <BSTree.BSTree instance at 0x7f09fde945f0>,
<BSTree.BSTree instance at 0x7f09fde94d40>, <BSTree.BSTree instance at 0x7f09fdc71f80>,
<BSTree.BSTree instance at 0x7f09fdc71368>, <BSTree.BSTree instance at 0x7f09fdc715a8>,
<BSTree.BSTree instance at 0x7f09fdc71ab8>])
```

The above call estimates the probability of a BST being a list by generating 100 random BSTs each of which has 5 nodes with random keys in `[0, 1000]` and then computing the percentage of these BSTs that are linear. The first number indicates that of 100 random BSTs only 13 BSTs are lists. Obviously, the percentage of linear BSTs may be different in repeated calls of this function, because the BSTs are randomly generated. Use the following function to repeatedly estimate the probabilities of BSTs being lists. The first two arguments specify the range of the number of nodes in the tree, the third argument is the number of random BSTs to generate for each number of nodes. The last two arguments, `a` and `b`, specify the range in which random keys are generated. The method returns a dictionary that maps each number of nodes to the output of `estimate_list_prob_in_rand_bsts_with_num_nodes()`.

```
def estimate_list_probs_in_rand_bsts(num_nodes_start, num_nodes_end, num_trees, a, b):
    d = {}
    for num_nodes in xrange(num_nodes_start, num_nodes_end+1):
        d[num_nodes] = estimate_list_prob_in_rand_bsts_with_num_nodes(num_trees, num_nodes, a, b)
    return d
```

Estimate the probabilities by the following call:

```
>> d = estimate_list_probs_in_rand_bsts(5, 200, 1000, 0, 1000000)
```

In other words, for each number of nodes in `[5, 200]`, generate 1000 random BSTs with keys randomly chosen from `[0, 1000000]` and compute the percentages of those BSTs that are lists. Below is how we can print the estimated probabilities. Since we are dealing with random numbers, your floats will most likely be different.

```
>>> for k, v in d.iteritems():
      print('probability of linearity in rbsts with %d nodes = %f' % (k, v[0]))

probability of linearity in rbsts with 5 nodes = 0.122000
probability of linearity in rbsts with 6 nodes = 0.035000
probability of linearity in rbsts with 7 nodes = 0.011000
probability of linearity in rbst with 8 nodes = 0.001000
probability of linearity in rbsts with 9 nodes = 0.001000
...
```

On to the function

```
estimate_balance_prob_in_rand_bsts_with_num_nodes(num_trees, num_nodes, a, b).
```

This function behaves in the exact same way as `estimate_list_prob_in_rand_bsts_with_num_nodes` but returns the 2-tuple where the first number is the probability of BST being balanced. You can test your implementation with the following function that builds a dictionary of probabilities.

```
def estimate_balance_probs_in_rand_bsts(num_nodes_start, num_nodes_end, num_trees, a, b):
    d = {}
    for num_nodes in xrange(num_nodes_start, num_nodes_end+1):
        d[num_nodes] = estimate_balance_prob_in_rand_bsts_with_num_nodes(num_trees, num_nodes, a, b)
    return d
```

Here is a test run.

```
>>> d = estimate_balance_probs_in_rand_bsts(5, 200, 1000, 0, 1000000)
>>> for k, v in d.iteritems():
print('probability of balance in rbsts with %d nodes = %f' % (k, v[0]))
for k, v in d.iteritems():
              print('probability of balance in rbsts with %d nodes = %f' % (k, v[0]))

probability of balance in rbsts with 5 nodes = 0.329000
probability of balance in rbsts with 6 nodes = 0.129000
probability of balance in rbsts with 7 nodes = 0.153000
probability of balance in rbsts with 8 nodes = 0.129000
probability of balance in rbsts with 9 nodes = 0.101000
probability of balance in rbsts with 10 nodes = 0.049000
probability of balance in rbsts with 11 nodes = 0.017000
probability of balance in rbsts with 12 nodes = 0.024000
probability of balance in rbsts with 13 nodes = 0.017000
probability of balance in rbsts with 14 nodes = 0.011000
probability of balance in rbsts with 15 nodes = 0.014000
probability of balance in rbsts with 16 nodes = 0.011000
probability of balance in rbsts with 17 nodes = 0.004000
probability of balance in rbsts with 18 nodes = 0.000000
probability of balance in rbsts with 19 nodes = 0.002000
probability of balance in rbsts with 20 nodes = 0.001000
probability of balance in rbsts with 21 nodes = 0.004000
probability of balance in rbsts with 22 nodes = 0.000000
probability of balance in rbsts with 23 nodes = 0.001000
...
```

Note that that since we are dealing with random numbers, your output will be slightly different.

## 10 Plotting Conditional Probabilities

Let's plot these probabilities. Toward that end, implement the function

```
plot_rbst_lin_probs(num_nodes_start, num_nodes_end, num_trees)
```

The first two numbers, `num_nodes_start` and `num_nodes_end`, specify the range of the number of nodes in random binary search trees and `num_trees` specifies the number of random trees that must be generated for each number of nodes in the range. When I called `plot_rbst_lin_probs(1, 50, 1000)`, I got the graph shown in Figure 1.

Implement the function

```
plot_rbst_balance_probs(num_nodes_start, num_nodes_end, num_trees)
```

The first two numbers, `num_nodes_start` and `num_nodes_end`, specify the range of the number of nodes in random BSTs and `num_trees` specifies the number of random BSTs that must be generated for each number of nodes in the range. When I called `plot_rbst_balance_probs(1, 50, 1000)`, I got the graph shown in Figure 2.

## 11 What To Submit

The zip for this assignment contains `BSTNode.py`, `BSTree.py`, `rand_bst.py`. You shouldn't have to modify anything in `BSTNode.py`. The file `rand_bst.py` has the starter code with the functions you need to define for this assignment. Define them and submit your `BSTree.py`, `rand_bst.py`, and `BSTNode.py` via Canvas. At the beginning of `rand_bst.py`, add brief answers to the following two questions:

1. As the number of nodes in a binary search tree goes to infinity, what is the probability of a binary search tree being a list?
2. As the number of nodes in a binary search tree goes to infinity, what is the probability of a binary search tree being balanced?

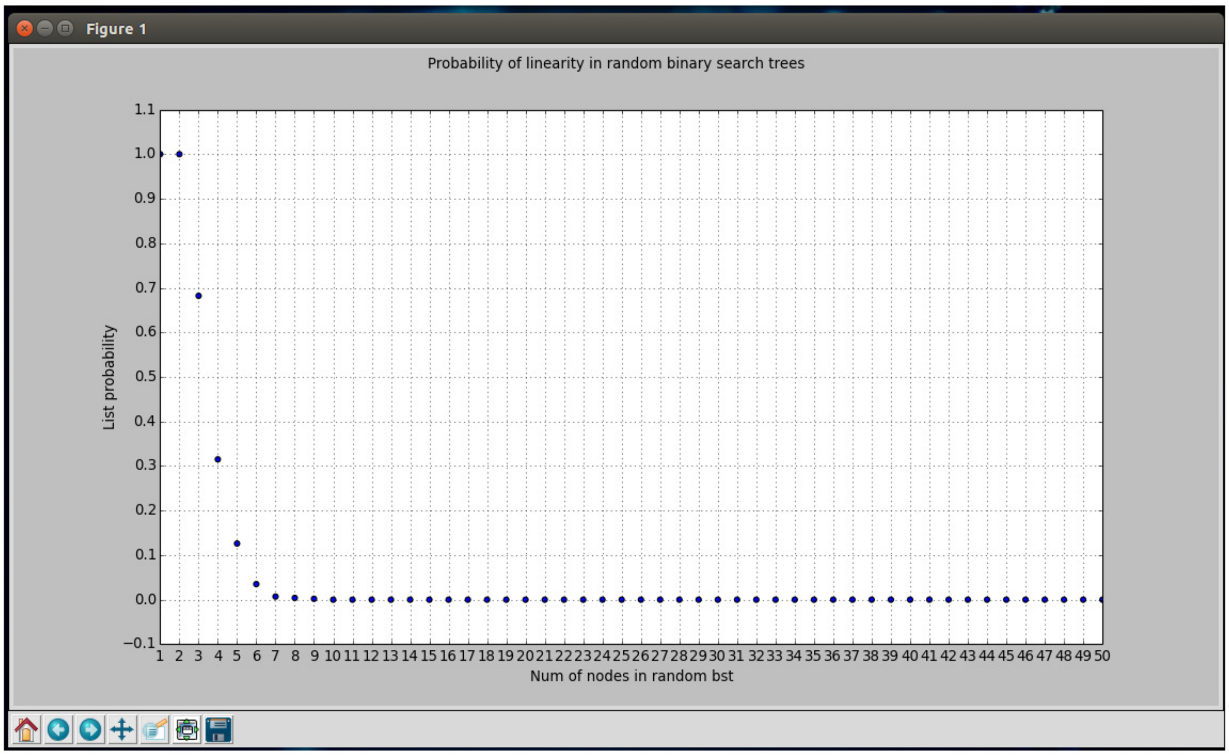

Figure 1: Probability of random binary search trees being lists

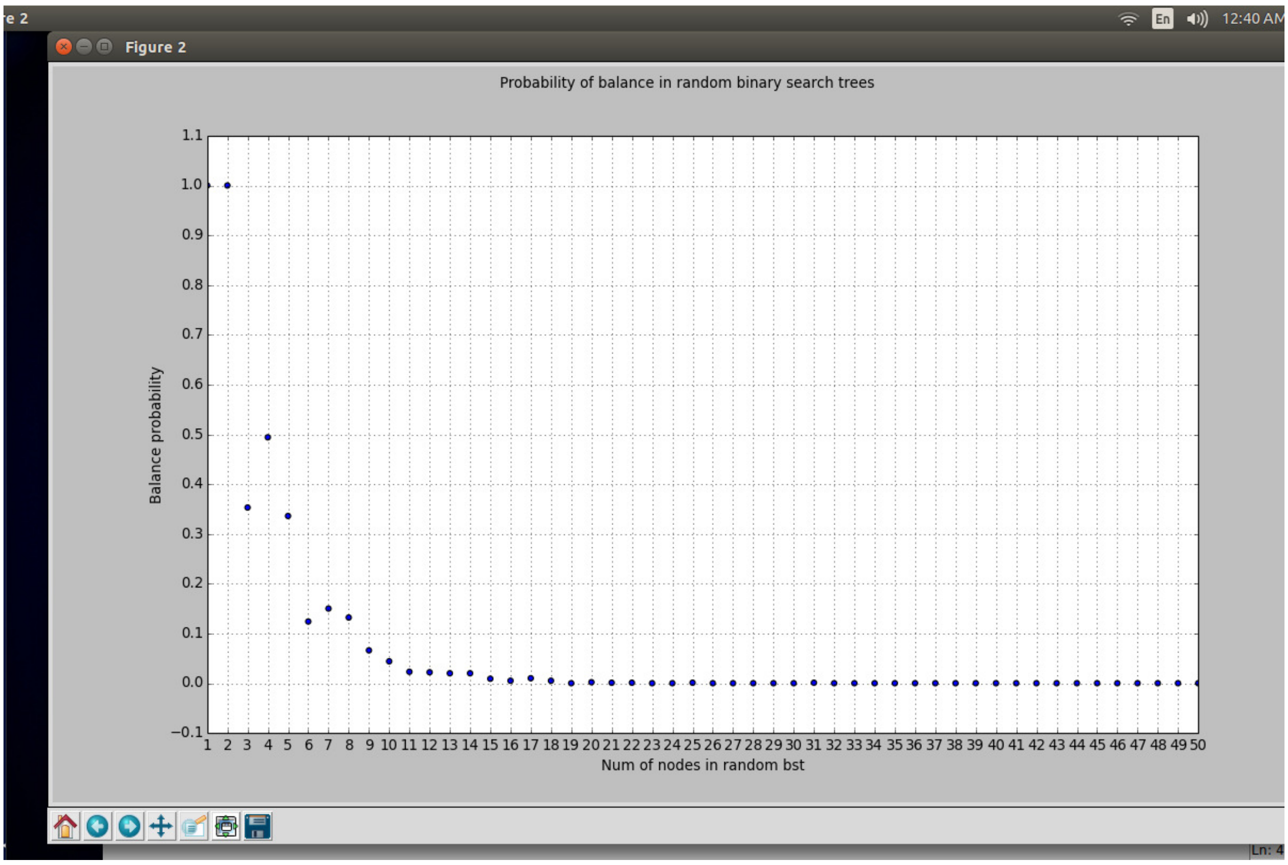

Figure 2: Probability of random binary search trees being balanced

Happy Hacking!