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

March 25, 2017

1 Learning Objectives

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

2 Introduction

In this assignment, you will learn (or review if you have already studied it) one of the most useful data structures in computer science - the binary search tree. We will discuss some theory behind binary search trees, implement a few methods of the binary search tree class and then use them to estimate the probability of a randomly generated binary search tree being a list, i.e., linear, or balanced. The techniques of estimating this likelihood will serve as a gentle introduction to conditional probability. This assignment will also give you another chance of playing with Matplotlib and OOP in Python.

3 Binary Search Tree

A binary search tree is a tree (possibly empty, i.e., without any nodes) where each node has at most two child nodes (hence the term `binary`) such that 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.

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

The trees above are all binary search trees. Each node has at most two children and for each node, the keys to the left of it are strictly less whereas the keys to the right of it are strictly greater. In the second tree from the left, for example, key 5 is to the left of key 10. In the third tree from the left, $5 < 10$ and $20 > 10$, etc. Note that this condition must hold true for `every` node in the tree. Thus, the following tree is a binary tree but not a binary search tree:

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

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

4 Binary Search Tree Nodes

A binary search tree consists of binary search tree nodes. Each node contains a key. This key is typically an integer. A blueprint of the binary search tree 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:

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

We just constructed a binary tree search node `bn1` with a key of 10. When we apply `str()` to it, it 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 us 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=+`. This simply means that this node has both children. 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 Binary Search Tree

The file `BSTree.py` has a blueprint of the binary search tree class. As the constructor of the `BSTree` class shows, a `BSTree` object has two attributes: the root node (possibly empty) 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 binary search trees allow for the presence of duplicates. For the sake of simplicity, in this assignment we will deal with binary search trees with no duplicates. This method returns `True` when the node is successfully inserted and `False` if the node's key is already in the tree. If the tree has no nodes, the key becomes the root's key. Otherwise, we drill down the tree comparing the key to the key of the current node. If the key is less, we set the current node to its left child and keep going down. 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 it. Let us create a binary search tree 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 is what `displayInOrder` prints out when called on 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
```

`NULL`s are displayed for empty nodes. The above tree in bst can be drawn as follows, 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 binary search tree node is defined recursively. The height of a `NULL` node (`None` in Py jargon) is -1. The height of a binary search tree node with a key but not children is 0. Otherwise, the height of a binary search tree node is 1 + max(height of the node's left child, height of the node's right child). The height of a binary search tree is the height of its root. Let us get back to the BSTs discussed above.

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

Then going left to right, the height of the first tree is 0, the height of the 2 nd tree is 1, the height of the 3 rd tree is 1, the height of the 4 th tree is 2, and the height of the 5 th tree is 2 as well. A binary search tree is linear, i.e. a list, when the number of nodes in the tree is exactly 1 greater than its height.

In `BSTree.py`, you will need to implement the method `isList(self)` that returns `True` when the binary search tree is a list and `False` otherwise. You will also need to implment the method `heightOf(self)` that returns the height of the binary search tree. Here is an example:

```
>>> 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 binary search tree is balanced if for every node the heights of the left and right children differ but at most 1. 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. In `BSTree.py`, implement the method `isBalanced(self)` that returns `True` if the binary search tree is balanced and `False` otherwise.

8 Random Binary Search Trees

Now that we have `isList(self)` and `isBalanced(self)`, we can start experimenting with probabilities. Toward that end, implement a function `gen_rand_bst(num_nodes, a, b)`. The first argument specifies the number of nodes in a binary search tree, the second and third arguments specify the range of for the random number generator that generates integers and inserts them into the tree until the tree has the required number of nodes. The method returns a binary search tree object. Here is an example:

```
>>> 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
```

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 binary search tree being a list by generating `num_trees` random binary search trees, each of which has `num_nodes` of random numbers in `[a, b]`, counting the number of the binary search trees that are lists and dividing that number by `num_trees`. The function returns 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:

```
>>> 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 binary search tree being a list by generating 100 random binary search trees each of which has 5 nodes with random keys in `[0, 100]`. You can use the following function to repeatedly estimate the probabilities of binary search trees 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 binary search trees 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]`, we generate 1000 random binary search trees with keys randomly chosen from `[0, 1000000]` and compute the ratio of those trees 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 slightly 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 binary search tree 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
```

Below 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, you output will be slightly different.

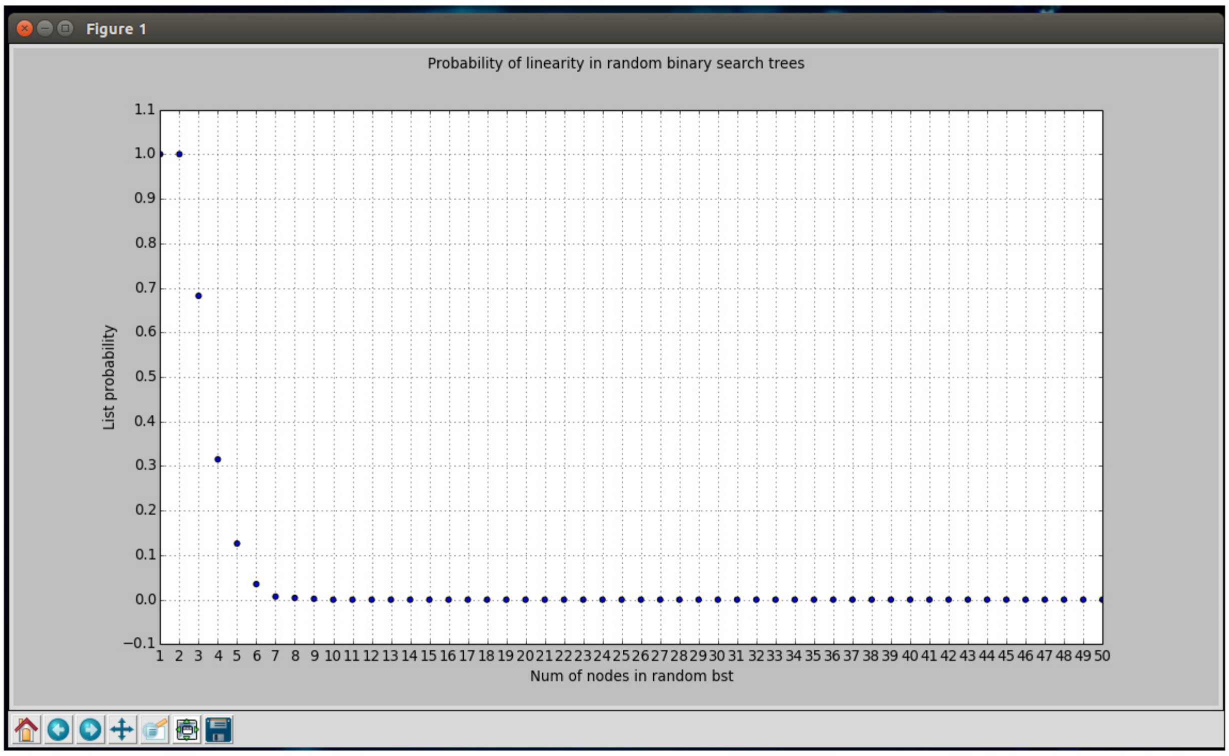

Figure 1: Probability of random binary search trees being lists

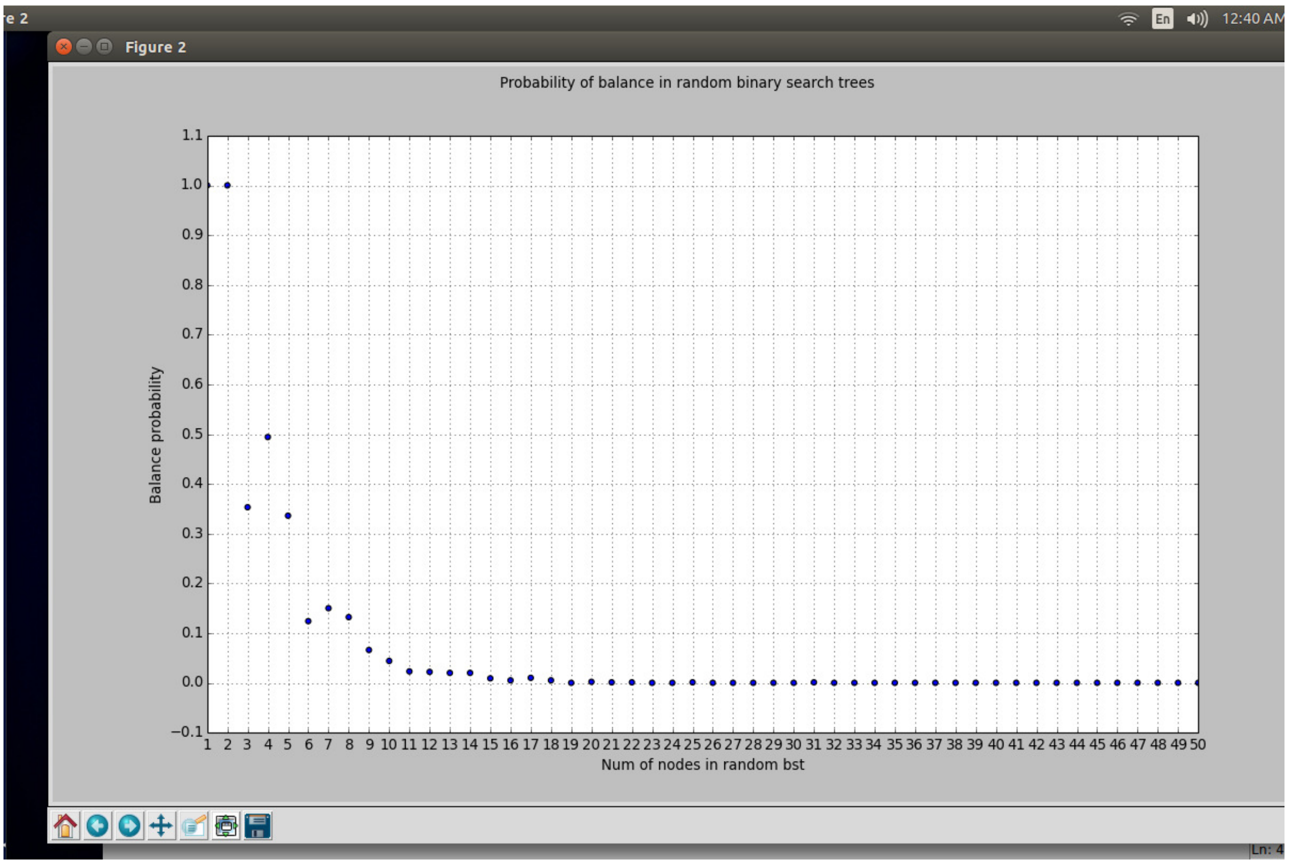

Figure 2: Probability of random binary search trees being balanced

10 Plotting Conditional Probabilities

Let us plot the computed 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. The actual numbers should be in `[0, 1000000]`. 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 binary search trees and `num_trees` specifies the number of random trees that must be generated for each number of nodes in the range. The actual numbers should be in `[0, 1000000]`. 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 do not 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 in a zip archive. At the beginning of `rand_bst.py`, add brief answers to the following three 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?
3. What probability distribution is the plot generated by your `plot_rbst_lin_probs` most similar to?
4. What probability distribution is the plot generated by your `plot_rbst_balance_probs` most similar to?

Happy Hacking!