Objectives

* Familiarity with Binary Search Trees
* Class inheritance
* Unittest
* Tree/node equality
* Use of DOCSTRING

Due Date

This assignment is due on Tuesday, April 2 by 11:00 pm.

**Remarks:**

* It is imperative you develop test cases for your code. (See structure in TestMyBSTMap.py provided)
* When you are asked to hand in code, you cut-and-paste the definition(s) from your code and paste into this Word document immediately following the activity where indicated.
* This homework is to be individual work.

Value

This assignment is worth 20 points.

Activities

1. In the .zip file provided, there are three files that you will need for this homework. Create a project in your development environment containing these three files:

* BSTMap.py – contains the class definition for a binary tree map. You do not need to modify this code. You will inherit this class in classes you build and then provide additional methods.
* TestMyBSTMap.py – is a template for unittests that you need to create. You should build out these tests BEFORE you do any coding. Making use of the provided BSTMap.py class definition, you can build these tests easily with the methods provided.
* MyBSTMap.py – is a programming template where you have to build up code for additional methods. This is a starting place for your code development. You will be handing in your full class definition when completed.

1. Verify your class definitions in MyBSTMap.py inherit the super class definitions in BSTMap.py.
2. OVERVIEW: You will be adding two pieces of functionality as outlined in MyBSTMap.py and shown in the figure below. Familiarize yourself with the code provided as reflected in the figure below:

* Equality - Return True if two trees share the same key:value pairs and shape
* Tree reconstruction - returns a BST equal to a tree used to create a list of (key:value) pairs via preorder or postorder traversal.



**Figure 1:** Class diagrams for this assignment. The dark-background boxes are the classes you will be coding. The light-background boxes are already complete.

1. Build out the DOCSTRING notation for your methods. The DOCSTRING should clearly provided:

* Syntax for the method call.
* Description of arguments (what is going into the method)
* Description of output (what is being modified in the object or returned from the method)

1. Build out the unittests for equality methods in the TestMyBSTMap.py template. Insights on how the methods should behave are provided in the following step. Include at a minimum the following unittests:

* 2 empty trees (should be equal)
* 2 equal trees with several levels of nodes
* 2 unequal trees with several layers of nodes:
  1. same key:value pairs, different shapes
  2. same shapes, different key:value pairs

An example/discussion later may provide some insights into how to structure these tests.

1. Build out the \_\_eq\_\_ methods in both class definitions MyBSTMap and MyBSTNode provided in the template MyBSTMap.py. This method should return True or False. The user will call equality on the two trees (bst1 == bst2). Most of the work here is recursively defining equality for a node - two nodes are equal if:

* They have the same key:value pairs
* The subtrees rooted at their left and right children are equal

|  |  |  |
| --- | --- | --- |
| 3:'3' |  | 3:'3' |
| / |  | / |
| 1:'1' | == | 1:'1' These trees are equal |
| / \ |  | / \ |
| 0:'0' 2:'2' |  | 0:'0' 2:'2' |
| 3:'3' |  | 1:'1' |
| / |  | / \ |
| 1:'1' | != | 0:'0' 3:'3' These trees are not equal. They have the same |
| / \ |  | / key:value pairs as content, but they are different |
| 0:'0' 2:'2' |  | 2:'2' structures |

1. (Read this entire step and the follow-on examples before executing) Build out two static methods in MyBSTMap.py. Several examples are provided in the follow-on discussion:

* frompreorder(L) – generates a BST given the preorder list of key:value pairs of the tree. It takes in a preorder list of key:value pairs and returns the generated BST. Note that BSTMap.py already provides a preorder method to provide a preorder listing of a BST.
* frompostorder(L) – provides the same functionality given a postorder list of key:value pairs.

Consider the following tree of key:value pairs:

|  |
| --- |
| 3:'3'  /  1:'1'  / \  0:'0' 2:'2' |

Verify for yourself that traversals of this tree yields the following lists of key:value tuples:

* pre-order: [(3,'3'), (1,'1'), (0,'0'), (2,'2')]
* post-order: [(0,'0'), (2,'2'), (1,'1'), (3,'3')]

Further note BSTMap.py provides supporting magic methods to print out a tree in a pseudo-tree like fashion (see the main program after the class definition)

**Examples:** Do NOT use these examples for your own tests. Create your own.

|  |
| --- |
| from MyBSTMap import MyBSTMap  bst1 **=** MyBSTMap()  **for** k **in** [3, 1, 2, 0]: *# Build the tree*  bst1.put(k, str(k))  L **=** [(k, v) **for** (k, v) **in** bst1.preorder()] *# construct preorder list*  bst2 **=** MyBSTMap.frompreorder(L) *# reconstruct the original bst*  bst1 **==** bst2 *# verify trees are equal*  **>>>** True |

**Unittests:** Start with a small tree (~3 levels), and include ascii art of the tree as above. Do not use any of the trees or shapes given in this assignment.

Then, compare two large random trees (at least 100 nodes)

1. create a tree using randomly generated keys (use the random module)
2. create a list of k:v pairs using preorder traversal
3. create a new tree using that list MyBSTMap.frompreorder()
4. verify the two trees are equal
5. modify one of the trees
6. verify the two trees are no longer equal.

Repeat the above for frompostorder.

1. Paste below each bullet your final code:

* MyBSTMap.py

from BSTMap import BSTMap, BSTNode # provided for you  
  
# Inherit from BSTMap, but overload `newnode` to use this one instead  
class MyBSTMap(BSTMap):  
   
 def newnode(self, key, value = None):   
 return MyBSTNode(key, value) # overloads the `newnode` method to use MyBSTNode() instead of BSTNode()  
  
 def \_\_eq\_\_(self, other):  
 *"""  
 Compares two MyBSTMap instances for equality.  
  
 Syntax: \_\_eq\_\_(self, other)  
 Arguments:  
 - other: Another MyBSTMap instance to compare with.  
  
 Output: True if the two MyBSTMap instances have identical tree structures  
 and corresponding key-value pairs, False otherwise.  
 """* # The heavy lifting here is done in the corresponding  
 # function in MyBSTNode - just tell it which node to  
 # start with.  
 if not isinstance(other, MyBSTMap):  
 return False  
 return self.root == other.root  
  
  
 # these are "static" methods - they belong to the class but do not take an instance of  
 # the class as a parameter (no `self``).  
 # note the "decorator" @staticmethod - this let's python know this is not a typical "bound" method  
 @staticmethod  
 def frompreorder(L):  
 *"""  
 Constructs a MyBSTMap from a given preorder sequence.  
  
 Syntax: frompreorder(L)  
 Arguments:  
 - L: A list of key-value tuples in preorder traversal order.  
  
 Output: A new MyBSTMap instance populated with the key-value pairs from the preorder sequence.  
 """* new\_bst = MyBSTMap()  
 for key\_value in L:  
 new\_bst.put(key\_value[0], key\_value[1])  
 return new\_bst  
  
 @staticmethod  
 def frompostorder(L):  
 *"""  
 Constructs a MyBSTMap from a given postorder sequence.  
  
 Syntax: frompostorder(L)  
 Arguments:  
 - L: A list of key-value tuples in postorder traversal order.  
  
 Output: A new MyBSTMap instance populated with the key-value pairs from the postorder sequence.  
 """* new\_bst = MyBSTMap()  
 for key\_value in reversed(L):  
 new\_bst.put(key\_value[0], key\_value[1])  
 return new\_bst  
  
  
class MyBSTNode(BSTNode):  
   
 newnode = MyBSTMap.newnode # overloads the `newnode` method to use the correct Node class  
  
 def \_\_eq\_\_(self, other):  
 *"""  
 Compares two MyBSTNode instances for equality.  
  
 Syntax: \_\_eq\_\_(self, other)  
 Arguments:  
 - other: Another MyBSTNode instance to compare with.  
  
 Output: True if the two MyBSTNode instances have the same key, value,  
 and subtree structure, False otherwise.  
 """* if other is None:  
 return False  
 if self.key != other.key or self.value != other.value:  
 return False  
 left\_eq = self.left == other.left if self.left and other.left else self.left is other.left  
 right\_eq = self.right == other.right if self.right and other.right else self.right is other.right  
 return left\_eq and right\_eq

* TestMyBSTMap.py (Start a new page for this bullet and the code)

import unittest, random  
from MyBSTMap import MyBSTMap  
  
  
class TestBSTMap(unittest.TestCase):  
 def test\_equal\_empty(self):  
 *"""  
 Tests the equality of two empty MyBSTMap instances.  
  
 Syntax: test\_equal\_empty()  
 Arguments: None  
 Output: Asserts that two empty MyBSTMap instances are equal.  
 """* bst1 = MyBSTMap()  
 bst2 = MyBSTMap()  
 self.assertEqual(bst1, bst2, "Two empty MyBSTMap instances should be equal.")  
  
 def test\_equal\_multiplenodes(self):  
 *"""  
 Tests the equality of two MyBSTMap instances with identical key-value pairs.  
  
 Syntax: test\_equal\_multiplenodes()  
 Arguments: None  
 Output: Asserts that two MyBSTMap instances with the same key-value pairs are equal.  
 """* # Create the first MyBSTMap instance and add multiple key-value pairs  
 bst1 = MyBSTMap()  
 bst1.put(3, 'Three')  
 bst1.put(1, 'One')  
 bst1.put(4, 'Four')  
 bst1.put(2, 'Two')  
  
 # Create the second MyBSTMap instance and add the same key-value pairs in the same order  
 bst2 = MyBSTMap()  
 bst2.put(3, 'Three')  
 bst2.put(1, 'One')  
 bst2.put(4, 'Four')  
 bst2.put(2, 'Two')  
  
 # Assert that the two instances are considered equal  
 self.assertEqual(bst1, bst2, "Two MyBSTMap instances with identical nodes should be equal.")  
  
 def test\_notequal\_multiplenodes\_difshapes(self):  
 *"""  
 Tests the inequality of two MyBSTMap instances with the same keys and values but different structures.  
  
 Syntax: test\_notequal\_multiplenodes\_difshapes()  
 Arguments: None  
 Output: Asserts that two structurally different MyBSTMap instances are not equal.  
 """* # Create the first MyBSTMap instance and add nodes in a specific order  
 bst1 = MyBSTMap()  
 bst1.put(4, 'Four')  
 bst1.put(2, 'Two') # Adding in a way that '2' becomes a left child  
 bst1.put(6, 'Six')  
 bst1.put(1, 'One') # Further nesting to define a specific tree shape  
  
 # Create the second MyBSTMap instance and add the same nodes but in a different order  
 bst2 = MyBSTMap()  
 bst2.put(4, 'Four')  
 bst2.put(6, 'Six') # Adding in a way that '6' is directly a right child  
 bst2.put(2, 'Two') # Changing the structure compared to bst1  
  
 # Assert that the two instances are considered not equal  
 self.assertNotEqual(bst1, bst2,  
 "Two MyBSTMap instances with identical nodes but different structures should not be equal.")  
  
 def test\_notequal\_multiplenodes\_difkvs(self):  
 *"""  
 Tests the inequality of two MyBSTMap instances with different key-value pairs.  
  
 Syntax: test\_notequal\_multiplenodes\_difkvs()  
 Arguments: None  
 Output: Asserts that two MyBSTMap instances with different key-value pairs are not equal.  
 """* bst1 = MyBSTMap()  
 bst1.put(1, 'One')  
  
 bst2 = MyBSTMap()  
 bst2.put(1, 'Two') # same key but different value compared to bst1  
  
 self.assertNotEqual(bst1, bst2, "Two MyBSTMap instances with different key-value pairs should not be equal.")  
  
 def test\_frompreorder\_small(self):  
 *"""  
 Tests constructing a small MyBSTMap instance from a preorder sequence.  
  
 Syntax: test\_frompreorder\_small()  
 Arguments: None  
 Output: Validates the MyBSTMap instance constructed from a small preorder sequence.  
 """* preorder\_list = [(4, 'Four'), (2, 'Two'), (1, 'One'), (3, 'Three'), (5, 'Five')]  
  
 # Use this preorder sequence to create a MyBSTMap instance  
 bst\_from\_preorder = MyBSTMap.frompreorder(preorder\_list)  
  
 # Now create a similar tree manually to compare  
 expected\_bst = MyBSTMap()  
 expected\_bst.put(4, 'Four')  
 expected\_bst.put(2, 'Two')  
 expected\_bst.put(1, 'One')  
 expected\_bst.put(3, 'Three')  
 expected\_bst.put(5, 'Five')  
  
 # Assert that the BST created from the preorder sequence is equal to the manually created BST  
 self.assertEqual(bst\_from\_preorder, expected\_bst,  
 "The BST constructed from the provided preorder sequence does not match the expected BST.")  
  
 def test\_frompreorder\_large(self):  
 *"""  
 Tests constructing a large MyBSTMap instance from a preorder sequence.  
  
 Syntax: test\_frompreorder\_large()  
 Arguments: None  
 Output: Validates the MyBSTMap instance constructed from a large preorder sequence.  
 """* preorder\_list = [(50, 'Fifty'), (30, 'Thirty'), (20, 'Twenty'),  
 (40, 'Forty'), (70, 'Seventy'), (60, 'Sixty'),  
 (80, 'Eighty'), (75, 'Seventy Five'), (90, 'Ninety')]  
  
 # Use this preorder sequence to create a MyBSTMap instance  
 bst\_from\_preorder = MyBSTMap.frompreorder(preorder\_list)  
  
 # Now create a similar tree manually to compare  
 expected\_bst = MyBSTMap()  
 for kv\_pair in preorder\_list:  
 expected\_bst.put(kv\_pair[0], kv\_pair[1])  
  
 # Assert that the BST created from the preorder sequence is equal to the manually created BST  
 self.assertEqual(bst\_from\_preorder, expected\_bst,  
 "The BST constructed from the large preorder sequence does not match the expected BST.")  
  
 def test\_frompostorder\_small(self):  
 *"""  
 Tests constructing a small MyBSTMap instance from a postorder sequence.  
  
 Syntax: test\_frompostorder\_small()  
 Arguments: None  
 Output: Validates the MyBSTMap instance constructed from a small postorder sequence.  
 """* postorder\_list = [(1, 'One'), (3, 'Three'), (2, 'Two'), (5, 'Five'), (4, 'Four')]  
  
 bst\_from\_postorder = MyBSTMap.frompostorder(postorder\_list)  
  
 expected\_bst = MyBSTMap()  
 for kv in [(4, 'Four'), (2, 'Two'), (1, 'One'), (3, 'Three'),  
 (5, 'Five')]: # This preorder sequence would result in the same tree as the postorder list above  
 expected\_bst.put(kv[0], kv[1])  
  
 self.assertEqual(bst\_from\_postorder, expected\_bst,  
 "The BST constructed from the provided small postorder sequence does not match the expected BST.")  
  
 def test\_frompostorder\_large(self):  
 *"""  
 Tests constructing a large MyBSTMap instance from a postorder sequence.  
  
 Syntax: test\_frompostorder\_large()  
 Arguments: None  
 Output: Validates the MyBSTMap instance constructed from a large postorder sequence.  
 """* postorder\_list = [(20, 'Twenty'), (40, 'Forty'), (30, 'Thirty'), (60, 'Sixty'), (90, 'Ninety'), (80, 'Eighty'),  
 (70, 'Seventy'), (50, 'Fifty')]  
  
 bst\_from\_postorder = MyBSTMap.frompostorder(postorder\_list)  
  
 expected\_bst = MyBSTMap()  
 for kv in [(50, 'Fifty'), (30, 'Thirty'), (20, 'Twenty'), (40, 'Forty'), (70, 'Seventy'), (60, 'Sixty'),  
 (80, 'Eighty'), (90, 'Ninety')]: # A preorder sequence that would build the same tree  
 expected\_bst.put(kv[0], kv[1])  
  
 self.assertEqual(bst\_from\_postorder, expected\_bst,  
 "The BST constructed from the provided large postorder sequence does not match the expected BST.")  
  
 def test\_small\_tree\_ascii(self):  
 *"""  
 Tests printing out a small tree that follows the BST's ascii representation  
  
 Syntax: test\_small\_tree\_ascii()  
 Description of Arguments: None  
 Description of Output: Displays a small tree with ASCII art but does not perform any actual assertion.  
 """* print(" Small Tree (ASCII Art):")  
 print(" 2 ")  
 print(" / \\ ")  
 print(" 1 3 ")  
  
 def test\_large\_random\_trees\_preorder(self):  
 *"""  
 Syntax: test\_large\_random\_trees\_preorder()  
 Description of Arguments: None  
 Description of Output: Tests creating, comparing, modifying, and re-comparing large BSTs based on preorder traversal.  
 """* bst1 = MyBSTMap()  
 num\_nodes = 100  
 random\_keys = [random.randint(1, 1000) for \_ in range(num\_nodes)]  
 for key in random\_keys:  
 bst1.put(key, str(key)) # Using key as value  
 preorder\_list = [(key, value) for key, value in bst1.preorder()]  
  
 bst2 = MyBSTMap.frompreorder(preorder\_list)  
  
 self.assertEqual(bst1, bst2, "The two BSTs should be equal after creation from preorder.")  
  
 bst1.put(-1, "-1") # Add a new node to bst1  
 self.assertNotEqual(bst1, bst2, "The two BSTs should not be equal after modification.")  
  
 def test\_large\_random\_trees\_postorder(self):  
 *"""  
 Syntax: test\_large\_random\_trees\_postorder()  
 Description of Arguments: None  
 Description of Output: Tests creating, comparing, modifying, and re-comparing large BSTs based on postorder traversal.  
 """* bst1 = MyBSTMap()  
 num\_nodes = 100  
 random\_keys = [random.randint(1, 1000) for \_ in range(num\_nodes)]  
 for key in random\_keys:  
 bst1.put(key, str(key)) # Using key as value  
 postorder\_list = [(key, value) for key, value in bst1.postorder()]  
  
 bst2 = MyBSTMap.frompostorder(postorder\_list)  
  
 self.assertEqual(bst1, bst2, "The two BSTs should be equal after creation from postorder.")  
  
 bst1.put(-1, "-1") # Add a new node to bst1  
 self.assertNotEqual(bst1, bst2, "The two BSTs should not be equal after modification.")  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()