COMP 314: Algorithms and Complexity
Lab work 3: Binary Search Tree (using OOP)

1 Purpose

Implementation and testing of binary search tree.

2 Tasks

Students are required to accomplish the following tasks in Python.

1. Create a class called `BinarySearchTree`. Implement the following operations in this class:
   (a) `size()`: Return the number of nodes in the tree.
   (b) `add(key, value)` : Add a node to a BST.
   (c) `search(key)`: Search BST for the requested key and return the corresponding value. Must return `False` if the key does not exist.
   (d) `smallest()`: Find the smallest key and return the key-value pair (tuple).
   (e) `largest()`: Find the largest key and return the key-value pair (tuple).
   (f) `remove(key)`: Delete a key from a BST. Must return `False` if the key does not exist.
   (g) `inorder_walk()`: Inorder traversal. Must return a list of keys visited in inorder way.
   (h) `preorder_walk()`: Preorder traversal. Must return a list of keys visited in preorder way.
   (i) `postorder_walk()`: Postorder traversal. Must return a list of keys visited in postorder way.

   All your implementation must be in a file named `bst.py`. Class name and method names must be as specified above (case-sensitive).
2. Run the test cases provided in Section 4. These test cases are also provided in KULMS.
3. Write some other test cases to test further.

3 Readings

1. For binary search tree and algorithms:
   Chapter 14 of Necaise, R. D. (2010). Data Structures and Algorithms Using Python.
   Chapter 2 of Horowitz et al. (2013). Fundamentals of Computer Algorithms.
   Chapter 12 of Cormen et al. (2014). Introduction to Algorithms.
2. For unit testings: `https://docs.python.org/3/library/unittest.html`
3. For OOP in Python: `https://realpython.com/python3-object-oriented-programming/`

4 Test cases

```
import unittest
from bst import BinarySearchTree

class BSTTestCase(unittest.TestCase):

    def setUp(self):
        """
        Executed before each test method.
        Before each test method, create a BST with some fixed key-values.
        """
        self.bst = BinarySearchTree()
        self.bst.add(10, "Value for 10")
        self.bst.add(52, "Value for 52")
        self.bst.add(5, "Value for 5")
        self.bst.add(8, "Value for 8")
        self.bst.add(1, "Value for 1")
        self.bst.add(40, "Value for 40")
        self.bst.add(30, "Value for 30")
        self.bst.add(45, "Value for 45")

    def test_add(self):
        """
        tests for add
        """
        # Create an instance of BinarySearchTree
        bsTree = BinarySearchTree()

        # bsTree must be empty
        self.assertEqual(bsTree.size(), 0)

        # Add a key-value pair
        bsTree.add(15, "Value for 15")
        # Size of bsTree must be 1
        self.assertEqual(bsTree.size(), 1)

        # Add another key-value pair
        bsTree.add(10, "Value for 10")
        # Size of bsTree must be 2
        self.assertEqual(bsTree.size(), 2)

        # The added keys must exist.
        self.assertEqual(bsTree.search(10), "Value for 10")
        self.assertEqual(bsTree.search(15), "Value for 15")

    def test_inorder(self):
        """
        tests for inorder_walk
        """
        actual_output = self.bst.inorder_walk()
        expected_output = [1, 5, 8, 10, 30, 40, 45, 52]

        self.assertListEqual(actual_output, expected_output)

        # Add one node
        self.bst.add(25, "Value for 25")
        # Inorder traversal must return a different sequence
        self.assertListEqual(self.bst.inorder_walk(), [1, 5, 8, 10, 25, 30, 40, 45, 52])

    def test_postorder(self):
        """
        tests for postorder_walk
        """
        actual_output = self.bst.postorder_walk()
        expected_output = [1, 8, 5, 30, 45, 40, 52, 10]

        self.assertListEqual(actual_output, expected_output)

        # Add one node
```

```
        self.bst.add(25, "Value for 25")
        # Postorder traversal must return a different sequence
        self.assertListEqual(self.bst.postorder_walk(), [1, 8, 5, 25, 30, 45, 40, 52, 10])

    def test_preorder(self):
        """
        tests for preorder_walk
        """
        self.assertListEqual(self.bst.preorder_walk(), [10, 5, 1, 8, 52, 40, 30, 45])

        # Add one node
        self.bst.add(25, "Value for 25")
        # Preorder traversal must return a different sequence
        self.assertListEqual(self.bst.preorder_walk(), [10, 5, 1, 8, 52, 40, 30, 25, 45])

    def test_search(self):
        """
        tests for search
        """
        actual_output = self.bst.search(40)
        expected_output = "Value for 40"
        self.assertEqual(actual_output, expected_output)

        self.assertFalse(self.bst.search(90))

        self.bst.add(90, "Value for 90")
        self.assertEqual(self.bst.search(90), "Value for 90")

    def test_remove(self):
        """
        tests for remove
        """
        self.bst.remove(40)

        self.assertEqual(self.bst.size(), 7)
        self.assertListEqual(self.bst.inorder_walk(), [1, 5, 8, 10, 30, 45, 52])
        self.assertListEqual(self.bst.preorder_walk(), [10, 5, 1, 8, 52, 30, 45])

    def test_smallest(self):
        """
        tests for smallest
        """
        self.assertTupleEqual(self.bst.smallest(), (1, "Value for 1"))

        # Add some nodes
        self.bst.add(6, "Value for 6")
        self.bst.add(4, "Value for 4")
        self.bst.add(0, "Value for 0")
        self.bst.add(32, "Value for 32")

        # Now the smallest key is 0.
        self.assertTupleEqual(self.bst.smallest(), (0, "Value for 0"))

    def test_largest(self):
        """
        tests for largest
        """
        self.assertTupleEqual(self.bst.largest(), (52, "Value for 52"))

        # Add some nodes
        self.bst.add(6, "Value for 6")
        self.bst.add(54, "Value for 54")
        self.bst.add(0, "Value for 0")
        self.bst.add(32, "Value for 32")

        # Now the largest key is 54
        self.assertTupleEqual(self.bst.largest(), (54, "Value for 54"))

if __name__ == "__main__":
    unittest.main()
```