Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab Report 3

COMP 314

(For partial fulfillment of 3rd Year/ 2nd Semester in Computer Engineering)

Submitted to:

Dr. Rajani Chulyadyo

Department of Computer Science and Engineering

Submitted by:

Neha Malla

C.E.

Roll No. 27

1. Source Code

➔ Filename: bst.py

```
class BinarySearchTree:
    def __init__(self):
        self.BSTsize = 0
        self.root = None

    class BSTnode:
        def __init__(self, key, value):
            self.key = key
            self.value = value
            self.left = None
            self.right = None

    # Add a node to the BST
    def add(self, key, value):
        newNode = self.BSTnode(key, value)
        self.BSTsize += 1

        if self.root == None:
            self.root = newNode
        else:
            node = self.root
            while node != None:
                if key <= node.key:
                    if node.left == None:
                        node.left = newNode
                        break
                    else:
                        node = node.left
                else:
                    if node.right == None:
                        node.right = newNode
                        break
                    else:
```

```
                        node = node.right

    # Return the number of nodes in the BST
    def size(self):
        return self.BSTsize

     # Perform inorder traversal. Must return a list of keys visited in
inorder way, e.g. [1, 2, 3, 4].
    def inorder_walk(self):
        return self.inorder(self.root, [])

    def inorder(self, root, inList):
        if root != None:
            self.inorder(root.left, inList)
            inList.append(root.key)
            self.inorder(root.right, inList)
        return inList

     # Perform postorder traversal. Must return a list of keys visited in
inorder way, e.g. [1, 4, 3, 2].
    def postorder_walk(self):
        return self.postorder(self.root, [])

    def postorder(self, root, postList):
        if root != None:
            self.postorder(root.left, postList)
            self.postorder(root.right, postList)
            postList.append(root.key)
        return postList

     # Perform preorder traversal. Must return a list of keys visited in
inorder way, e.g. [2, 1, 3, 4].
    def preorder_walk(self):
        return self.preorder(self.root, [])

    def preorder(self, root, preList):
```

```
        if root != None:
            preList.append(root.key)
            self.preorder(root.left, preList)
            self.preorder(root.right, preList)
        return preList

    # Search the BST for the given key. Return False if the key is not
found.
    def search(self, key):
        root = self.root
        while root != None:
            if key == root.key:
                return root.value
            elif key < root.key:
                root =  root.left
            else:
                root = root.right
        return False

    # Remove a key from the BST. Return False if the key is not present in
the BST.
    def remove(self, key):
        # root: node to be deleted, element: node to be swapped with root
        root = self.root

        while root != None:# searching the node
            if key == root.key:
                break
            parent = root# parent of the node to be removed
            if key < root.key:
                root =  root.left
                child = "left"
            else:
                root = root.right
                child = "right"
```

```
        if root != None:
            self.BSTsize -= 1
             if root.left == None and root.right == None:# node is a leaf
node
                if self.root == root:
                    self.root = None
                elif child == "left":
                    parent.left = None
                elif child == "right":
                    parent.right = None

            elif root.left == None and root.right != None:# node has right
subtree only
                if self.root == root:
                    self.root = root.right
                elif child == "left":
                    parent.left = root.right
                elif child == "right":
                    parent.right = root.right

              elif root.left != None and root.right == None:# node has a
left subtree only
                if self.root == root:
                    self.root = root.left
                elif child == "left":
                    parent.left = root.left
                elif child == "right":
                    parent.right = root.left

            else:# node has both children
                   # finding 'element' that is the largest node from left
subtree of the node to be removed
                element = root.left
                while element.right != None:
                    parentElement = element# parent of element
                    element = element.right
```

```
                #replacing the node with element
                root.key = element.key
                root.value = element.value

                    if root.left == element:# largest element in the left
subtree is the left child of the node itself
                    root.left = element.left
                else:
                     if element.left == None:# largest element node has no
left subtree
                        parentElement.right = None
                    else:
                        parentElement.right = element.left
        else:
            return False

      # Find the smallest key and return the corresponding key-value
pair/tuple, i.e. (key, value)
    def smallest(self):
        root = self.root
        while root.left != None:
            root = root.left
        return root.key, root.value

      # Find the largest key and return the corresponding key-value
pair/tuple, i.e. (key, value)
    def largest(self):
        root = self.root
        while root.right != None:
            root = root.right
        return root.key, root.value
```

2. Test Cases

➔ Filename: test_bst.py

```
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
        """
          self.assertListEqual(self.bst.inorder_walk(), [1, 5, 8, 10, 30,
40, 45, 52])

        # 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
        """
         self.assertListEqual(self.bst.postorder_walk(), [1, 8, 5, 30, 45,
40, 52, 10])

        # Add one node
        self.bst.add(25, "Value for 25")
        # Inorder 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")
        # Inorder 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
        """
        self.assertEqual(self.bst.search(40), "Value for 40")

        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])
```

```
        self.bst.remove(10)

        self.assertEqual(self.bst.size(), 6)
          self.assertListEqual(self.bst.inorder_walk(), [1, 5, 8, 30, 45,
52])
          self.assertListEqual(self.bst.preorder_walk(), [8, 5, 1, 52, 30,
45])

        self.bst.remove(52)

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

3. Output of Test Cases

```
C:\Users\neha\Algorhythm\lab3>python test_bst.py
........
----------------------------------------------------------------------
Ran 8 tests in 0.002s

OK

C:\Users\neha\Algorhythm\lab3>
```

4. Conclusion

Binary Search Tree was implemented using Object Oriented Programming. All the methods were iterative except for the inorder, preorder and postorder traversal methods which are recursive.

Since the remove method was a little complex, some test cases were added to validate the various cases that occured while removing a node. All the other given tests cases were enough, thus are unchanged.

All test cases ran successfully on the nine methods, as seen on the screenshot provided above.