Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Algorithms and Complexity (COMP 314)

Lab III

Submitted To:

Dr. Rajani Chulyadyo

Department of Computer Science and Engineering

Submitted By:

Mission Shrestha (54)

Year/Sem: III/II

Submission Date: 2nd May, 2023

Implementation and testing of binary search tree.

1. **Create a class called BinarySearchTree and Implement the following operations in this class :**

   I. **size():**

```python
class BinarySearchTree:

    def __init__(self, key=0, value=0):
        self.root = None
        self._size = 0

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

 # Return the number of nodes in the BST

    def size(self):
        return self._size
```

II. add() :

```python
# Add a node to the BST

    def add(self, key, value):
        z = self.BSTNode(key, value)
        y = None
        x = self.root

        while (x != None):
            y = x
            if (z.key < x.key):
                x = x.left
            else:
                x = x.right

        if (y == None):
            self.root = z
        elif (z.key < y.key):
            y.left = z
        else:
            y.right = z

        self._size += 1
```

III. search():

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

IV. smallest():

```python
# Find the smallest key and return the corresponding key-value pair

    def smallest(self):
        x = self.root
        while x.left != None:
            x = x.left

        return (x.key, x.value)

```

V. largest():

```python
# Find the largest key and return the corresponding key-value pair/tupl

    def largest(self):
        x = self.root
        while x.right != None:
            x = x.right

        return (x.key, x.value)

```

VI. remove():

```python
# Remove a key from the BST. Return False if the key is not present in the BST.
    def remove(self, key):
        x = self.search(key)
        if not x:
            return -1

        to_delete = self.root
        parent = None
        while (to_delete.key != key):
            parent = to_delete
            if (key < to_delete.key):
                to_delete = to_delete.left
            else:
                to_delete = to_delete.right

        #  leaf node case
        if (to_delete.right == None and to_delete.left == None):
            if parent.left == to_delete:
                parent.left = None
            else:
                parent.right = None
            self._size -= 1

        # one child case
        if (to_delete.left == None and to_delete.right != None) or (to_delete.right == None and to_delete.left != None):
            if (to_delete.left == None):
                to_replace = to_delete.right
                to_delete.right = None
            else:
                to_replace = to_delete.left
                to_delete.left = None

            to_delete.key = to_replace.key
            to_delete.value = to_replace.value
            self._size -= 1

        #  two children case
        if (to_delete.right != None and to_delete.left != None):
            to_replace = to_delete.left
            to_replace_parent = None

            if to_replace.right == None:
                to_delete.key = to_replace.key
                to_delete.value = to_replace.value
                to_delete.left = None
                self._size -= 1

            else:
                while (to_replace.right != None):
                    to_replace_parent = to_replace
                    to_replace = to_replace.right
                to_replace_parent.right = None
                to_delete.key = to_replace.key
                to_delete.value = to_replace.value
                self._size -= 1
```

VII. inorder walk():

```python
# Perform inorder traversal. Must return a list of keys visited in inorde

    def inorder_walk(self):
        stack = []
        list = []
        x = self.root

        while stack or x:
            if x:
                stack.append(x)
                x = x.left
            else:
                x = stack.pop()
                list.append(x.key)
                x = x.right

        return list

```

VIII. preorder walk():

```python
# Perform preorder traversal. Must return a list of keys visited in i
    def preorder_walk(self):
        x = self.root

        stack = []
        stack.append(x)

        list = []
        while stack:
            x = stack.pop()
            list.append(x.key)
            if x.right:
                stack.append(x.right)
            if x.left:
                stack.append(x.left)

        return list
```

IX. postorder walk():

```
# Perform postorder traversal. Must return a list of keys visited in inorder way, e.g. [1, 4, 3, 2].

    def postorder_walk(self):
        x = self.root
        stack = []
        stack.append(x)
        list = []

        while stack:
            x = stack.pop()
            list.append(x.key)

            if x.left:
                stack.append(x.left)

            if x.right:
                stack.append(x.right)

        list = list[::-1]

        return list
```

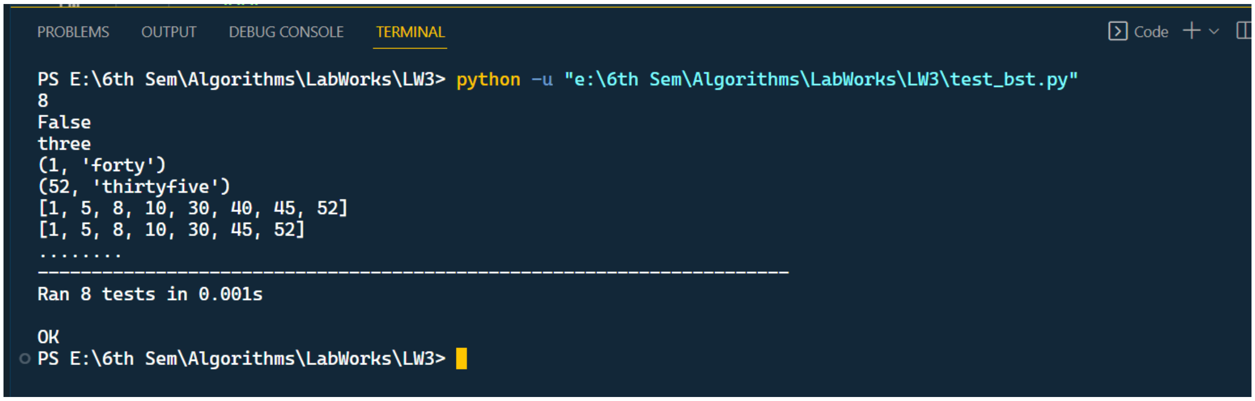
```python

tree = BinarySearchTree()


tree.add(10, "ten")
tree.add(52, 'thirtyfive')
tree.add(5, "five")
tree.add(8, "twenty")
tree.add(1, "forty")
tree.add(40, "three")
tree.add(30, "six")
tree.add(45, "fifteen")


print(tree.size())
print(tree.search(13))
print(tree.search(40))
print(tree.smallest())
print(tree.largest())


# print(tree.preorder_walk())
# tree.remove(40)
# print(tree.preorder_walk())

print(tree.inorder_walk())
tree.remove(40)
print(tree.inorder_walk())

# print(tree.postorder_walk())
# tree.remove(40)
# print(tree.postorder_walk())

```

```
PS E:\6th Sem\Algorithms\LabWorks\LW3> python -u "e:\6th Sem\Algorithms\LabWorks\LW3\bst.py"
8
False
three
(1, 'forty')
(52, 'thirtyfive')
[1, 5, 8, 10, 30, 40, 45, 52]
[1, 5, 8, 10, 30, 45, 52]
PS E:\6th Sem\Algorithms\LabWorks\LW3>
```

2. Test cases to test your program.

```python
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()
```

When test cases were run:



Conclusion:

The Binary search tree with eight different functions were implemented. The program was tested using test cases. The implemented binary search tree passed all of the given test cases.