Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Algorithms and Complexity (COMP 314)

Lab 3 Report

Submitted to:

Dr. Rajani Chulyadyo

Department of Computer Science and Engineering

Submitted By:

Mani Dumaru

Roll no.: 15

CE-2019 3rd year/2nd semester

Submission Date: 2nd May, 2023

Implementation and testing of Binary Search Tree

Source Code Link: binarySrarchTree

1. For adding a node to the binary search tree

```
def add(self, key, value):
    node = Node(key, value)

    if (self.root == None):
        self.root = node
        self.treeSize += 1
        # print(f"{node.key}: root node added")
        return node
    else:
        x = self.root
        while(x != None):
            y = x
            if (node.key < x.key):
                x = x.left
            else:
                x = x.right
        if(node.key < y.key):
            y.left = node
            self.treeSize += 1
            # print(f"{node.key} added to left of {y.key}")
            return
        else:
            y.right = node
            self.treeSize += 1
            # print(f"{node.key} added to right of {y.key}")
            return
```

2. Finding the size of the tree

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

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

3. For searching a key in the tree

```
def search(self, key):
    if (self.root == None):
        return -1
    else:
        x = self.root
        while(x != None):
            if (x.key == key):
                # print(f"{key} found with value {x.value}")
                return x.value
            elif(key < x.key):
                x = x.left
            else:
                x = x.right
    # print(f"{key} not found")
    return False
```

4. For finding the smallest key in the binary search tree

```
def smallest(self):
    if(self.root == None):
        return -1
    else:
        x = self.root
        while(x != None):
            y = x
            x = x.left
        result = (y.key, y.value)
        return result
```

5. For finding the largest key in the binary search tree

```
def largest(self):
    if(self.root == None):
        return -1
    else:
        x = self.root
        while(x != None):
            y = x
            x = x.right
        result = (y.key, y.value)
        return result
```

6. Inorder Walk in the tree

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

def inorder_walk(self):
    walk = []
    self.inorder(walk, self.root)
    return walk
```

7. Preorder Walk in the tree

```
def preorder_walk(self):
    walk = []
    self.preorder(walk, self.root)
    return walk

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

8. Postorder walk in the tree

```
def postorder_walk(self):
    walk = []
    self.postorder(walk, self.root)
    return walk

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

9. Removing an element from the tree

```
def remove(self, key):
    x = self.search(key)
    if not x:
        # print("key not found to delete")
        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

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

    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.treeSize -= 1

    if (to_delete.right != None and to_delete.left != None): ################### if two children ####################
        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.treeSize -= 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.treeSize -= 1
```

Result for Test Cases:

```
D:\CE-2019\Sem 6\lab works\Algorithm\lab3>python test.py
........
----------------------------------------------------------------------
Ran 8 tests in 0.002s

OK

D:\CE-2019\Sem 6\lab works\Algorithm\lab3>
```

Conclusion

Hence, given test cases were used to test the correctness of the above algorithms. All test cases passed the correctness of the algorithms. In this way a binary search tree with 8 different functions was implemented.