Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A lab Report 3

On

"Algorithm and Complexity"

[Course Code: COMP 314]

Submitted By:

Sabil Shrestha (51)

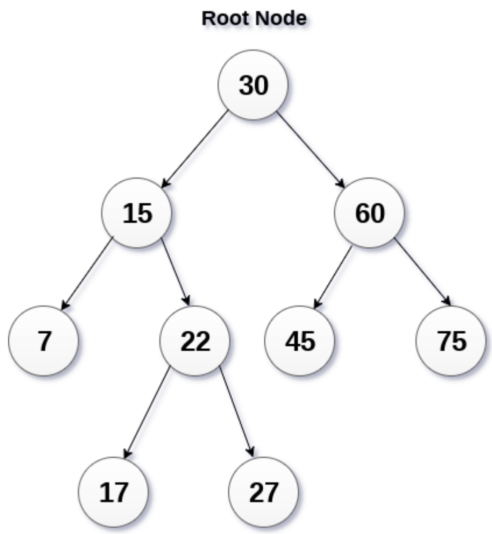
Submitted To:

Rajani Chulyadyo Mam

Submission Date: 14th November, 2020

Binary Search Tree

Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree. In a Binary search tree, the value of all the nodes in the left sub-tree is less than the value of root. value of all the nodes in the right sub-tree is greater than or equal to the value of the root. The information represented by each node is a record rather than a single data element. One of the advantages of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient. The binary search tree is efficient data structure if compared with arrays and linked lists. BST removes half sub-tree at every step. So, it is very fast and efficient. Searching for an element in a binary search tree takes o(log2n) time. In worst case, the time it takes to search an element is 0(n) which is in linear time and efficient.



Binary Search Tree

While inserting value in BST the left part should always be less than the parent and the right part should always be greater than the parent node. If this rule isn’t followed then it is not a Binary Search Tree. After deletion of value from a node. If the node is leaf node or a node with no children then no other operation should be performed to maintain the Binary Search Tree. But if the node isn’t a leaf node or node with children then either the largest element from left most subtree should be selected as new value of the node or the smallest element from right most node is selected. The insertion of values in BST, searching of BST, removal of value from BST and making of BST after the removal of value are some operations.

Operations in a Binary Search Tree:

1. Insertion

While inserting value in BST the left part should always be less than the parent and the right part should always be greater than the parent node

```python
# Add a node to the BST
def add(self, key, value):
    if self.root is None:
        self.root = self._BinaryTreeNode(key, value)
    else:
        stree =self.root
        while True:
            if(key < stree.key):
                if(stree.left is not None):
                    stree = stree.left
                else:
                    stree.left = self._BinaryTreeNode(key,value)
                    break
            else:
                if(stree.right is not None):
                    stree = stree.right
                else:
                    stree.right = self._BinaryTreeNode(key,value)
                    break
    self._size +=1
```

2. In order

In inoder traversal first left node is selected than parent and then the right node.

```python
# Perform inorder traversal. Must return a list of keys v
def inorder_walk(self):
    found = []
    self._inorder(self.root,found)
    return found

def _inorder(self, stree, found):
    if(stree is not None):
        self._inorder(stree.left,found)
        found.append(stree.key)
        self._inorder(stree.right,found)
    return found
```

3. Postorder

In Postorder traversal first parent is selected than left and then right.

```
# Perform postorder traversal. Must return a list of keys visited in 
def postorder_walk(self):
    found = []
    self._postorder_walk(self.root,found)
    return found

def _postorder_walk(self, stree, found ):
    if(stree):
        self._postorder_walk(stree.left, found)
        self._postorder_walk(stree.right, found)
        found.append(stree.key)
```

4. Preorder

Preorder traversal is similar in concept with DFS traversal. First the left node is selected then right node and then the parent is selected.

```
# Perform preorder traversal. Must return a list of keys visited in inor
def preorder_walk(self):
    found = []
    self._preorder_walk(self.root, found)
    return found

def _preorder_walk(self,stree,found):
    if(stree):
        found.append(stree.key)
        self._preorder_walk(stree.left, found)
        self._preorder_walk(stree.right, found)
```

5. Search

The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root

```
# Search the BST for the given key. Return False if the key is not 
def search(self, key):
    sroot = self.root
    while sroot is not None:
        if (key == sroot.key):  #if direct node then return its val
            return sroot.value
        elif (key < sroot.key):     #if the key is less then root t
            sroot = sroot.left
        else:                       #if key is greater then root th
            sroot = sroot.right
    return False                        #else no value
```

6. Deletion

After deletion of value from a node. If the node is leaf node or a node with no children then no other operation should be performed to maintain the Binary Search Tree.

```
# Remove a key from the BST. Return False if the key is not present
def remove(self, key):
    stree = self.root
    previous = self.root

    while stree is not None:
        if stree.key == key:    #when found

            if stree.left is None and stree.right is None: #end nod
                stree = None        #delete node here bst will be b
            else:

                if stree.right is None:    #if no right node only
                    if key < previous.key:
                        previous.left = stree.left
                        stree = None
                    else:
                        previous.right = stree.left
                        stree = None

                else:
                    #we have to make a correct binary tree if we de
                    # we can do by two methods either greatest valu
                    # or smallest value of right, here we do smalle
                    # because it is relatively easier
                    temp = stree.right
                    temp_previous = stree

                    while temp.left is not None: #until left node i
                        temp_previous = temp
                        temp = temp.left
```

```
                    stree.key = temp.key
                    stree.value = temp.value

                    if(temp.key < temp_previous.key):
                        temp_previous.left = None

                    else:
                        temp_previous.right = None

                    del temp
            self._size -= 1
            return True

        elif key < stree.key:
            previous = stree
            stree =  stree.left

        else:
            previous = stree
            stree = stree.right

    return False


# Find the smallest key and return the corresponding key-value pai
def smallest(self):
    stre = self.root

    while(stre.left is not None):
        stre = stre.left
```

re to search

7. Largest value

The largest value is in the rightmost node of the tree.

```
def largest(self):
       stree = self.root
       while(stree.right is not None): # until we dont find rightmo
st
           stree = stree.right        #keep on going right
        return(stree.key, stree.value)
```

8. Smallest value

The smallest value is in the leftmost part of the tree.

```
  def smallest(self):
       stre = self.root

       while(stre.left is not None):
           stre = stre.left
       return(stre.key,stre.value)
```

Source Code
(bst.py)

```
class BinarySearchTree:
    #initializing the class
    def __init__(self):
        self.root=None    #initially no root node
        self._size=0

    class _BinaryTreeNode:
        def __init__(self,key,value):
            self.left = None  #initially no nodes
            self.right = None
            self.key = key
            self.value=value

    # Add a node to the BST
    def add(self, key, value):
        if self.root is None:
            self.root = self._BinaryTreeNode(key, value)
        else:
            stree =self.root
            while True:
                if(key < stree.key):
                    if(stree.left is not None):
                        stree = stree.left
                    else:
                        stree.left = self._BinaryTreeNode(key,value)
                        break
                else:
                    if(stree.right is not None):
                        stree = stree.right
                    else:
                        stree.right = self._BinaryTreeNode(key,value)
                        break
        self._size +=1
        # nod = self._BinaryTreeNode(key,value)
        # store = None
```

```
        # rot = self.root

        # while(rot != None):
        #     store = rot
        #     if(key < rot.key):
        #         rot = rot.left
        #     else:
        #         rot = rot.right
        # if (store == None):         #initially when we have to store value
        #     self.root = nod
        # elif (nod.key < store.key):
        #     store.left = rot
        # else:
        #     store.right = rot
        # self._size += 1


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

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

    def _inorder(self, stree, found):
        if(stree is not None):
            self._inorder(stree.left,found)
            found.append(stree.key)
            self._inorder(stree.right,found)
        return found

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

```
    def postorder_walk(self):
        found = []
        self._postorder_walk(self.root,found)
        return found

    def _postorder_walk(self, stree, found ):
        if(stree):
            self._postorder_walk(stree.left, found)
            self._postorder_walk(stree.right, found)
            found.append(stree.key)

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

    def _preorder_walk(self,stree,found):
        if(stree):
            found.append(stree.key)
            self._preorder_walk(stree.left, found)
            self._preorder_walk(stree.right, found)

    # Search the BST for the given key. Return False if the key is not found.
    def search(self, key):
        sroot = self.root
        while sroot is not None:
            if (key == sroot.key):  #if direct node then return its value
                return sroot.value
            elif (key < sroot.key):     #if the key is less then root then search lef
t subtree
                sroot = sroot.left
            else:                       #if key is greater then root than search righ
tsubtree
                sroot = sroot.right
        return False                        #else no value
```

```
    # Remove a key from the BST. Return False if the key is not present in the BST.
    def remove(self, key):
        stree = self.root
        previous = self.root

        while stree is not None:
            if stree.key == key:    #when found

                if stree.left is None and stree.right is None: #end node or leaf node
s
                    stree = None        #delete node here bst will be bst if we delet
e leaf nodes(no change in property)
                else:

                    if stree.right is None:    #if no right node only
                        if key < previous.key:
                            previous.left = stree.left
                            stree = None
                        else:
                            previous.right = stree.left
                            stree = None

                    else:
                        #we have to make a correct binary tree if we delete a node va
lue from midlle
                        # we can do by two methods either greatest value of left
                        # or smallest value of right, here we do smallest value of ri
ght
                        # because it is relatively easier
                        temp = stree.right
                        temp_previous = stree

                        while temp.left is not None: #until left node is not finished
                            temp_previous = temp
                            temp = temp.left
```

```
                        stree.key = temp.key
                        stree.value = temp.value

                        if(temp.key < temp_previous.key):
                            temp_previous.left = None

                        else:
                            temp_previous.right = None

                        del temp
                self._size -= 1
                return True

            elif key < stree.key:
                previous = stree
                stree =  stree.left

            else:
                previous = stree
                stree = stree.right

        return False


    # Find the smallest key and return the corresponding key-
value pair/tuple, i.e. (key, value)
    def smallest(self):
        stre = self.root

        while(stre.left is not None):
            stre = stre.left
        return(stre.key,stre.value)

    #     found =[]
    #     self._smallest(self.root,found)
    #     return found
```

```
    # def _smallest(self,stree, found):
    #     if(stree):
    #         if(stree.left == None):#the smallest element lies in left most
    #             found.append(stree.key)
    #         self._smallest(stree.left, found) #continue going at left side

    # Find the largest key and return the corresponding key-
value pair/tuple, i.e. (key, value)
    def largest(self):
        stree = self.root
        while(stree.right is not None): # until we dont find rightmost
            stree = stree.right        #keep on going right
        return(stree.key, stree.value)
```

Test case(test_bst.py)

```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
        """
        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, 5
2])

    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, 45, 30])

    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")
```

```python
        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()
```

Output

```
IPython 7.16.1 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/shres/Desktop/0 Study materials/0 This year/Algorithm and
Complexity/Lab/Lab 3/test_bst.py', wdir='C:/Users/shres/Desktop/0 Study materials/0
This year/Algorithm and Complexity/Lab/Lab 3')
........
----------------------------------------------------------------------
Ran 8 tests in 0.012s

OK

In [2]:
```

Observation

The operations in Binary Search Tree was observed. The insertion of values in BST, searching of BST, removal of value from BST and making of BST after the removal of value was performed using python. While inserting in BST the left part should always be less then the parent and the right part should always be greater than the parent node. If this rule isn't followed then it is not a Binary Search Tree. After deletion of value from a node. If the node is leaf node or a node with no children then no other operation should be performed to maintain the Binary Search Tree. But if the node isn't a leaf node or node with children then either the largest element from left most subtree should be selected as new value of the node or the smallest element from right most node is selected. The smallest value is present in the leftmost node of the tree and the largest value is present in the rightmost part of the tree. Traversal of the tree was performed. In order, preorder and post order traversal was performed in the binary search tree. Inorder traversal gives nodes in non-decreasing order. Preorder traversal is used to create a copy of the tree. Postorder traversal is used to delete the tree. In inorder traversal first left node is selected then parent and then right node is selected. In preorder traversal first the leftmost node is selected then right node and then the parent is selected. This preorder traversal is similar in concept to DFS traversal. In postorder traversal, parent is selected then left and right. The program was written in bst.py. The program was then tested with program test_bst.py. The test_bst.py was given by lecturer to test the algorithm we have written. All the tests were successfully completed using unitttest in python.