KATHMANDU UNIVERSITY

DHULIKHEL, KAVREPALANCHOWK, NEPAL

SUBJECT CODE: COMP 314

In the partial fulfilment of "Data Structures Revisited: Binary Search Tree"

Lab Report #3

<u>Submitted To:</u>

Rajani Chulyado,PhD
Lecturer
Department of Computer Science and Engineering (DoCSE)

<u>Submitted By:</u>

Abhay Raut
Roll No.:43
3rd Year, 6th Semester
B.E. Computer Engineering

Submission Date : 6th June 2019

Github Link : Abhay Raut

Introduction:

Binary Search Tree:

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
  There must be no duplicate nodes.

Basic Operations:

Following are the basic operations of a tree −

- **Search** − Searches an element in a tree.
- **Insert** − Inserts an element in a tree.
- **Pre-order Traversal** − Traverses a tree in a pre-order manner.
- **In-order Traversal** − Traverses a tree in an in-order manner.
- **Post-order Traversal** − Traverses a tree in a post-order manner.

Inserting a node

A naïve algorithm for inserting a node into a BST is that, we start from the root node, if the node to insert is less than the root, we go to left child, and otherwise we go to the right child of the root. We continue this process (each node is a root for some sub tree) until we find a null pointer (or leaf node) where we cannot go any further. We then insert the node as a left or right child of the leaf node based on node is less or greater than the leaf node. We note that a new node is always inserted as a leaf node. A recursive algorithm for inserting a node into a BST is as follows. Assume we insert a node N to tree T. if the tree is empty, the we return new node N as the tree. Otherwise, the problem of inserting is reduced to inserting the node N to left of right sub trees of T, depending on N is less or greater than T. A definition is as follows.

Insert(N, T) = N   if T is empty
= insert(N, T.left) if N < T
= insert(N, T.right) if N > T

Searching for a node

Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node). A recursive definition of search is as follows. If the node is equal to root, then we return true. If the root is null, then we return false. Otherwise we recursively solve the problem for T.left or T.right, depending on N < T or N > T. A recursive definition is as follows.

Search should return a true or false, depending on the node is found or not.
Search(N, T) = false   if T is empty
= true   if T = N
= search(N, T.left) if N < T
= search(N, T.right) if N > T

Deleting a node

A BST is a connected structure. That is, all nodes in a tree are connected to some other Node. For example, each node has a parent, unless node is the root. Therefore deleting a Node could affect all sub trees of that node. We need to be careful about deleting nodes from a tree. The best way to deal with deletion seems to be considering special cases. What if the node to delete is a leaf node? What if the node is a node with just one child? What if the node is an internal node (with two children). The latter case is the hardest to resolve. But we will find a way to handle this situation as well.

Case 1: The node to delete is a leaf node
This is a very easy case. Just delete the node. We are done

Case 2: The node to delete is a node with one child.

This is also not too bad. If the node to be deleted is a left child of the parent, then we connect the left pointer of the parent (of the deleted node) to the single child. Otherwise if the node to be deleted is a right child of the parent, then we connect the right pointer of the parent (of the deleted node) to single child.

Case 3: The node to delete is a node with two children

This is a difficult case as we need to deal with two sub trees. But we find an easy way to handle it. First we find a replacement node (from leaf node or nodes with one child) for the node to be deleted. We need to do this while maintaining the BST order property. Then we swap leaf node or node with one child with the node to be deleted (swap the data) and delete the leaf node or node with one child (case 1 or case 2)

Next problem is finding a replacement leaf node for the node to be deleted. We can easily find this as follows. If the node to be deleted is N, the find the largest node in the left sub tree of N or the smallest node in the right sub tree of N. These are two candidates that can replace the node to be deleted without losing the order property.

Applications include:

- Using linear data structures to represent sets: insert, isMember, remove, all O(n)

- Using binary search trees to represent sets: insert, isMember (search), remove, all O(h) --- O(lg(n)) if we are lucky.

Python Code:

```python
# Binary search Tree with function to add, search, delete , find largest,
# find lowest, inoredr, preoredr and postorder.

class BST:
    def __init__(self):
        self.size = 0  # as the initial size of the tree is zero
        self._root = None  # no elememt means no root element

    class BSTnode:
        def __init__(self, key, value):  # key is the value of node
            self.left = None
            self.right = None
            self.value = value  # value value
            self.key = key  # id of the single value

    # Add a node to a BST
    def add(self, key, value):
        z = self.BSTnode(key, value)
        x = self._root
        y = None
        while (x != None):  # when there is a root value
            y = x
            if (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

    # Getting the size of the tree
    def size_(self):
        return self.size

    # Search BST for the requested key
    def search(self, key):
        found = []
        self._search(self._root, key, found)
        return found

                self._search(subtree.left, key, found)
            elif (key > subtree.key):
                self._search(subtree.right, key, found)

    # Find the smallest key
    def SmallestKey(self):
        nodes = []
        self._SmallestKey(self._root, nodes)
        return nodes

    def _SmallestKey(self, subtree, nodes):
        if (subtree):
            if (subtree.left == None):
                nodes.append(subtree.key)
            self._SmallestKey(subtree.left, nodes)

    # Find the largest key
    def LargestKey(self):
        nodes = []
        self._LargestKey(self._root, nodes)
        return nodes

    def _LargestKey(self, subtree, nodes):
        if (subtree):
            if (subtree.right == None):
                nodes.append(subtree.key)
            self._LargestKey(subtree.right, nodes)

    # Delete a key from a BST
    # Given a binary search tree and a key, this function
    # delete the key and returns the new root
    def delete(self, key):
        found = []
        self._delete(self._root, key)
        return found

    def _delete(self, subtree, key):

        if subtree is None:
            return subtree

        # If the key to be deleted is smaller than the subtree's
        if key < subtree.key:
            subtree.left = self._delete(subtree.left, key)

        # If the kye to be delete is greater than the subtree's key
        elif (key > subtree.key):
            subtree.right = self._delete(subtree.right, key)
```

```python

        # If key is same as subtree's key, then this is the node
        else:
            # Node with only one child or no child
            if subtree.left is None:
                temp = subtree.right
                subtree = None
                return temp

            elif subtree.right is None:
                temp = subtree.left
                subtree = None
                return temp

            # Node with two children: Get the inorder successor
            temp = self.SmallestKey(subtree.right)
            # Copy the inorder successor's content to this node
            subtree.key = temp
            # Delete the inorder successor
            subtree.right = self._delete(subtree.right, temp)

        return subtree

    # In-order traversal
    def inorder(self):
        node = []
        self.inorder_walk(self._root, node)
        return node

    def inorder_walk(self, subtree, node):
        if subtree:
            self.inorder_walk(subtree.left, node)
            node.append(subtree.key)
            self.inorder_walk(subtree.right, node)

    # Pre-order traversal
    def preorder(self):
        node = []
        self.preorder_walk(self._root, node)
        return node

    def preorder_walk(self, subtree, node):
        if subtree:
            node.append(subtree.key)
            self.preorder_walk(subtree.left, node)
            self.preorder_walk(subtree.right, node)

    # Post-order traversal
    def postorder(self):
        node = []
        self.postorder_walk(self._root, node)
        return node

    def postorder_walk(self, subtree, node):
        if subtree:
            self.postorder_walk(subtree.left, node)
            self.postorder_walk(subtree.right, node)
            node.append(subtree.key)


if __name__ == "__main__":
    B = BST()
    n = int(input("Enter the no. of element in Search Tree: "))
    for x in range(n):
        value = int(input("Enter your element "))
        print('Adding ', value)
        B.add(value, "A value")

    print("Final Size is ", str(B.size))
    print()

    print("Largest Key", B.LargestKey())
    print("Smallest Key", B.SmallestKey())
    print()

    print("Inorder Sequence : ", B.inorder())
    print("Postorder Sequence : ", B.postorder())
    print("Preorder Sequence : ", B.preorder())
    print()

    _search = int(input("Enter the element to search Search Tree: "))
    print("Key Found in index: ", B.search(_search))
    print()

    _delete = int(input("Enter the element to delete Search Tree: "))
    print("Deleting key ", _delete, str(B.delete(_delete)))
    print("Inorder after deletion", B.inorder())
    print("Final Size is ", str(B.size))

```

Test Cases:

```python
from BinaryST import BST
import unittest

class BSTTestCase(unittest.TestCase):

    def test_bstTest(self):
        bst=BST()

        #Test Case for add
        bst.add(10,"Value for A")

        self.assertEqual(bst.size_(),1)         #Check for size. TRUE

        bst.add(5,"Value for B")
        self.assertEqual(bst.size_(),2)     #Check for size. TRUE

        bst.add(15,"Value for C")
        self.assertEqual(bst.size_(),3)     #Check for size. TRUE

        self.assertListEqual(bst.inorder(),[5,10,15])

        bst.add(11,"Value for D")           #Add value to check Traversal
        bst.add(20,"Value for E")           #Add value to check Traversal


        self.assertListEqual(bst.inorder(),[5,10,11,15,20])

        self.assertListEqual(bst.postorder(),[5,11,20,15,10])

        self.assertListEqual(bst.preorder(),[10,5,15,11,20])

        self.assertListEqual(bst.LargestKey(),[20])
        self.assertListEqual(bst.SmallestKey(),[5])
        self.assertListEqual(bst.search(20),[1])


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

Test Cases-Result:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
[Finished in 2.1s]
```

Conclusion:

Hence, several operations can be carried out in a binary search tree where the worst-case time of building a binary tree is O(n^2). On average, binary search trees with *n* nodes have O(log *n*) height. However, in the worst case, binary search trees can have O(*n*) height, when the unbalanced tree resembles a linked list(degenerate tree). So using binary search trees to represent sets is asymptotically no worse than lists, and often better. If we can find an efficient way to insure that our BSTs remain balanced then we can do asymptotically better than with lists.