

KATHMANDU UNIVERSITY

DHULIKHEL, KAVREPALANCHOWK, NEPAL



SUBJECT CODE: COMP 314

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

Lab Report #3

Submitted To:

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

Submitted By:

Kamal Shrestha
Roll No.: 49
6th Semester, 3rd Year
B.E. Computer Engineering

Submission Date: 27th May 2019

Link to Github: [https://github.com/KamalShrest/CE III 49 Lab3](https://github.com/KamalShrest/CE_III_49_Lab3)

Objectives:

1. Implement binary search tree. Your implementation must support the

Following operations:

- (a) Add a node to a BST
 - (b) Search BST for the requested key
 - (c) Find the smallest key
 - (d) Find the largest key
 - (e) Delete a key from a BST
 - (f) Inorder traversal
 - (g) Preorder traversal
 - (h) Postorder traversal
2. Write some test cases to test each operation listed above.

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, then we return new node N as the tree. Otherwise, the problem of inserting is reduced to inserting the node N to left or right sub trees of T, depending on N is less or greater than T. A definition is as follows.

$$\begin{aligned}\text{Insert}(N, T) &= N \quad \text{if } T \text{ is empty} \\ &= \text{insert}(N, T.\text{left}) \quad \text{if } N < T \\ &= \text{insert}(N, T.\text{right}) \quad \text{if } N > T\end{aligned}$$

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.

$$\begin{aligned}\text{Search}(N, T) &= \text{false} \quad \text{if } T \text{ is empty} \\ &= \text{true} \quad \text{if } T = N \\ &= \text{search}(N, T.\text{left}) \quad \text{if } N < T \\ &= \text{search}(N, T.\text{right}) \quad \text{if } N > T\end{aligned}$$

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, then 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:

```
1 # Binary Search Tree with function to add, search, delete, find largest,
2 # find lowest, inorder, preorder and postorder.
3
4 class BST:
5     def __init__(self):
6         self.size = 0 # as the initial size of the tree is zero
7         self._root = None # no element means no root element
8
9     class BSTnode:
10         def __init__(self, key, value): # key is the value of node
11             self.left = None
12             self.right = None
13             self.value = value # value value
14             self.key = key # id of the single value
15
16     # Add a node to a BST
17     def add(self, key, value):
18         z = self.BSTnode(key, value)
19         x = self._root
20         y = None
21         while (x != None): # when there is a root value
22             y = x
23             if (key < x.key):
24                 x = x.left
25             else:
26                 x = x.right
27         if (y == None):
28             self._root = z
29         elif (z.key < y.key):
30             y.left = z
31         else:
32             y.right = z
33         self.size += 1
34
35     # Getting the size of the tree
36     def size(self):
37         return self.size
38
39     # Search BST for the requested key
40     def search(self, key):
41         found = []
42         self._search(self._root, key, found)
43         return found
44
45     def _search(self, subtree, key, found):
46         if (subtree):
47             self._search(subtree.left, key, found)
48             if (key > subtree.key):
49                 self._search(subtree.right, key, found)
50
51     # Find the smallest key
52     def _SmallestKey(self):
53         nodes = []
54         self._SmallestKey(self._root, nodes)
55         return nodes
56
57     def _SmallestKey(self, subtree, nodes):
58         if (subtree):
59             if (subtree.left != None):
60                 nodes.append(subtree.key)
61                 self._SmallestKey(subtree.left, nodes)
62
63     # Find the largest key
64     def _LargestKey(self):
65         nodes = []
66         self._LargestKey(self._root, nodes)
67         return nodes
68
69     def _LargestKey(self, subtree, nodes):
70         if (subtree):
71             if (subtree.right != None):
72                 nodes.append(subtree.key)
73                 self._LargestKey(subtree.right, nodes)
74
75     # Delete a key from a BST
76     # Given a binary search tree and a key, this function
77     # delete the key and returns the new root
78     def delete(self, key):
79         found = []
80         self._delete(self._root, key)
81         return found
82
83     def _delete(self, subtree, key):
84         if subtree is None:
85             return subtree
86
87         # If the key to be deleted is smaller than the subtree's
88         # key
89         if key < subtree.key:
90             subtree.left = self._delete(subtree.left, key)
91
92         # If the key to be deleted is greater than the subtree's key
93         elif (key > subtree.key):
94             subtree.right = self._delete(subtree.right, key)
```

```

98
99     # If key is same as subtree's key, then this is the node
100 else:
101     # Node with only one child or no child
102     if subtree.left is None:
103         temp = subtree.right
104         subtree = None
105         return temp
106
107     elif subtree.right is None:
108         temp = subtree.left
109         subtree = None
110         return temp
111
112     # Node with two children: Get the inorder successor
113     temp = self.SmallestKey(subtree.right)
114     # Copy the inorder successor's content to this node
115     subtree.key = temp
116     # Delete the inorder successor
117     subtree.right = self._delete(subtree.right, temp)
118
119     return subtree
120
121 # In-order traversal
122 def inorder(self):
123     node = []
124     self.inorder_walk(self._root, node)
125     return node
126
127 def inorder_walk(self, subtree, node):
128     if subtree:
129         self.inorder_walk(subtree.left, node)
130         node.append(subtree.key)
131         self.inorder_walk(subtree.right, node)
132
133 # Pre-order traversal
134 def preorder(self):
135     node = []
136     self.preorder_walk(self._root, node)
137     return node
138
139 def preorder_walk(self, subtree, node):
140     if subtree:
141         node.append(subtree.key)
142         self.preorder_walk(subtree.left, node)
143         self.preorder_walk(subtree.right, node)
144

```

```

145 # Post-order traversal
146 def postorder(self):
147     node = []
148     self.postorder_walk(self._root, node)
149     return node
150
151 def postorder_walk(self, subtree, node):
152     if subtree:
153         self.postorder_walk(subtree.left, node)
154         self.postorder_walk(subtree.right, node)
155         node.append(subtree.key)
156
157
158 if __name__ == "__main__":
159     B = BST()
160     n = int(input("Enter the no. of element in Search Tree: "))
161     for x in range(n):
162         value = int(input("Enter your element "))
163         print('Adding ', value)
164         B.add(value, "A value")
165
166     print("Final Size is ", str(B.size))
167     print()
168
169     print("Largest Key", B.LargestKey())
170     print("Smallest Key", B.SmallestKey())
171     print()
172
173     print("Inorder Sequence : ", B.inorder())
174     print("Postorder Sequence : ", B.postorder())
175     print("Preorder Sequence : ", B.preorder())
176     print()
177
178     _search = int(input("Enter the element to search Search Tree: "))
179     print("Key Found in index: ", B.search(_search))
180     print()
181
182     _delete = int(input("Enter the element to delete Search Tree: "))
183     print("Deleting key ", _delete, str(B.delete(_delete)))
184     print("Inorder after deletion", B.inorder())
185     print("Final Size is ", str(B.size))
186

```

Test Cases:

```
1 from BinaryST import BST
2 import unittest
3
4 class BSTTestCase(unittest.TestCase):
5
6     def test_bstTest(self):
7         bst=BST()
8
9         #Test Case for add
10        bst.add(10,"Value for A")
11
12        self.assertEqual(bst.size_(),1)      #Check for size. TRUE
13
14        bst.add(5,"Value for B")
15        self.assertEqual(bst.size_(),2)      #Check for size. TRUE
16
17        bst.add(15,"Value for C")
18        self.assertEqual(bst.size_(),3)      #Check for size. TRUE
19
20        self.assertEqual(bst.inorder(),[5,10,15])
21
22        bst.add(11,"Value for D")      #Add value to check Traversal
23        bst.add(20,"Value for E")      #Add value to check Traversal
24
25
26        self.assertEqual(bst.inorder(),[5,10,11,15,20])
27
28        self.assertEqual(bst.postorder(),[5,11,20,15,10])
29
30        self.assertEqual(bst.preorder(),[10,5,15,11,20])
31
32        self.assertEqual(bst.LargestKey(),[20])
33        self.assertEqual(bst.SmallestKey(),[5])
34        self.assertEqual(bst.search(20),[1])
35
36
37 if __name__=="__main__":
38     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.