BinaryTree

February 27, 2017

```
In [1]: class TreeNode(object):
            def __init__(self):
                self.data = None
                self.left = None
                self.right = None
            def set_left(self, left_node):
                self.left = left_node
            def get_left(self):
                return self.left
            def get_right(self):
                return self.right
            def set_right(self, right_node):
                self.right = right_node
            def set_data(self, data):
                self.data = data
            def get_data(self):
                return self.data
```

```
In [146]: class BST(object):
              def __init__(self):
                  self.root = None
              # for insertion the base case is trivial
              # For other cases we need to keep track of predecessor while traversing the tree
              # as per the vaule of the node being inserted
              # a special case is if the new node's data already exists in the tree
              def insert(self, node):
                  if node == None:
                      print "node passed is None"
                  node_data = node.get_data()
                  if self.root == None:
                      self.root = node
                  else:
                      curr = self.root
                      pred = None
                      while curr != None:
                          if node_data < curr.get_data():
                              pred = curr
```

```
                curr = curr.get_left()
            elif node_data > curr.get_data():
                pred = curr
                curr = curr.get_right()
            elif node_data == curr.get_data():
                node.set_left(curr.get_left())
                curr.set_left(node)
                return
        assert curr == None
        if pred.get_data() >= node_data:
            pred.set_left(node)
        else:
            pred.set_right(node)
# In pre order traversal, we keep pushing left, right of every node in a queue
# so that nodes are printed in the order they are visited in pre order
def display_preorder(self, node):
    queue = list()
    queue.append(node)
    while len(queue) > 0:
        curr = queue[0]
        print curr.get_data()
        queue = queue[1:]
        left = curr.get_left()
        if left:
            queue.append(curr.get_left())
        right = curr.get_right()
        if right:
            queue.append(curr.get_right())
# In post order we need to print the children before the self
# this means we need to keep track of the children that are already visited
# so we need a visited set and a stack to keep the intermediate nodes
# a stack would be popped if either both the children are None or both the childre
# are visited
def display_postorder(self, node):
    curr = node
    visited = set()
    stack = list()
    # curr gets visited for the first time means curr gets touched
    # if curr has a left NOT in visited, then visit left
    # if curr has right NOT in visited, then visit right
    # if curr has no left and no right, then print curr and add curr to visited
    # unstack and check if left is visited, if left is not visited, then visit lef
    while curr != None or len(stack) > 0:
        stack.append(curr)
        left = curr.get_left()
        right = curr.get_right()
        #Base condition: leaf node or a node with both children visited
        if ((left == None) or left in visited) and ((right == None) or right in vi
```

```
            visited.add(curr)
            print curr.get_data()
            if len(stack) > 1:
                stack.pop()
                curr = stack.pop()
                continue
            else:
                return
        if (left != None) and (left not in visited):
            curr = left
        if (right != None) and (right not in visited):
            curr = right


def display(self, node):
    curr = node
    if curr != None:
        self.display(curr.left)
        print curr.get_data()
        self.display(curr.right)

# in inorder traversal, we keep going left and keep pushing
# in the stack, when we hit extreme left then we pop and go right
# and treat the right node as the new tree being traversed in inorder
def display_non_recursive(self, node):
    stack = list()
    curr = node
    while True:
        while curr != None:
            stack.append(curr)
            curr = curr.get_left()
        if len(stack) == 0:
            return
        curr = stack.pop()
        print curr.get_data()
        curr = curr.get_right()
def display_nr2(self, node):
    stack = list()
    curr = node
    while True:
        while curr != None:
            stack.append(curr)
            curr = curr.get_left()
        if len(stack) == 0:
            return
        curr = stack.pop()
        print curr.get_data()
        curr = curr.get_right()
```

```
            def get_root(self):
                return self.root

            # height of an empty tree is -1, height of a tree with just one node is 0
            # height ::= number of edges connecting the root with the deepest leaf
            def height(self, root):
                if root == None:
                    return -1
                else:
                    return 1 + max(self.height(root.get_left()), self.height(root.get_right()))
```

```
In [147]: nodes = [2, 1, 3, 7, 2.5, 2.7, 2.9]
          #nodes = [1, 0, 2, 12, -3, -14, 14, 23, 21, 22, 2]
```

```
In [148]: bst = BST()
          for node_data in nodes:
              node = TreeNode()
              node.set_data(node_data)
              bst.insert(node)
```

```
In [149]: bst.display(bst.get_root())
```

```
1
2
2.5
2.7
2.9
3
7
```

```
In [150]: bst.display_non_recursive(bst.get_root())
```

```
1
2
2.5
2.7
2.9
3
7
```

```
In [151]: bst.display_nr2(bst.get_root())
```

```
1
2
2.5
2.7
```

```
2.9
3
7
```

```
In [152]: bst.display_preorder(bst.get_root())
```

```
2
1
3
2.5
7
2.7
2.9
```

```
In [153]: bst.display_postorder(bst.get_root())
```

```
7
2.9
2.7
2.5
3
1
2
```

```
In [154]: bst.height(bst.get_root())
```

```
Out[154]: 4
```