DataStructures_Project_1_Firefly

February 5, 2021

0.1 Team Infomartion

Group Name: *Firefly* **Name 1**: *Omar Alaa* **ID 1**: *120180022* **Name 2**: *Yousef Saad* **ID 2**: *120180002* **Name 3**: *Martin Ihab* **ID 3**: *120180004*

0.2 Circular Deque

Design your implementation of the circular double-ended queue (deque). Your implementation should support following operations:

- `CircularDeque(capacity)` Constructor, set the maximum size of the deque to be capacity.
- `insertFront()` Adds an item at the front of Deque. Return true if the operation is successful.
- `insertRear()` Adds an item at the rear of Deque. Return true if the operation is successful.
- `deleteFront()` Deletes an item from the front of Deque. Return true if the operation is successful.
- `deleteRear()` Deletes an item from the rear of Deque. Return true if the operation is successful.
- `getFront()` Gets the front item from the Deque. If the deque is empty, return `None`.
- `getRear()` Gets the last item from Deque. If the deque is empty, return `None`.
- `isEmpty()` Checks whether Deque is empty or not.
-

0.3 `isFull()` Checks whether Deque is full or not.

Example

```
myCircularDeque = CircularDeque(3)  # set the size to be 3
myCircularDeque.insertRear(1)       # return true
myCircularDeque.insertRear(2)       # return true
myCircularDeque.insertFront(3)      # return true
myCircularDeque.insertFront(4)      # return false, the queue is full
myCircularDeque.getRear()           # return 2
myCircularDeque.isFull()            # return true
myCircularDeque.deleteRear()        # return true
myCircularDeque.insertFront(4)      # return true
```

```
myCircularDeque.getFront()          # return 4
```

[18]:
```
class CircularDeque(object):

    def __init__(self, k):
        self.queue = list()
        self.size = k
        self.used = 0

    def insertFront(self, value):
        if self.used >= self.size:
            return False
        self.queue = [value]+self.queue
        self.used += 1
        return True

    def insertRear(self, value):
        if self.used >= self.size:
            return False
        self.queue.append(value)
        self.used += 1
        return True

    def deleteFront(self):
        if self.used == 0:
            return
        self.queue.pop(0)
        self.used -= 1
        return True

    def deleteRear(self):
        if self.used == 0:
            return
        self.queue.pop(-1)
        self.used -= 1
        return True

    def getFront(self):
        if self.used == 0:
            return -1
        return self.queue[0]

    def getRear(self):
        if self.used == 0:
            return -1
        return self.queue[-1]
```

```
    def isEmpty(self):
        if self.used == 0:
            return True
        else:
            return False

    def isFull(self):
        if self.used == self.size:
            return True
        else:
            return False

myCircularDeque = CircularDeque(3) # set the size to be 3
print(myCircularDeque.insertRear(1))       # return true
print(myCircularDeque.insertRear(2))       # return true
print(myCircularDeque.insertFront(3))      # return true
print(myCircularDeque.insertFront(4))      # return false, the queue is full
print(myCircularDeque.getRear())           # return 2
print(myCircularDeque.isFull())            # return true
print(myCircularDeque.deleteRear())     # return true
print(myCircularDeque.insertFront(4))      # return true
print(myCircularDeque.getFront())          # return 4
```

```
True
True
True
False
2
True
True
True
4
```

0.4 AVL Tree

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1. AVL tree got its name after its inventors Georgy Adelson-Velsky and Landis.

Balance factor of a node in an AVL tree *is the difference between the height of the left subtree and that of the right subtree of that node* (i.e., Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)). The self balancing property of an avl tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1. An insertion or deletion process can corrupt the self balancing property of an AVL tree (i.e, the balance factor can be $<$ -1 or $>$ +1). Hence, *following the normal procedures of insertion or deletion of any regular binary search tree, one has to check whether the self balancing property of the AVL tree still holds or not.* If the property is violated, one has to re-balance the tree using a process called **rotation**.

Rotations are 3 types: left, right and double rotations. Actually, a double rotation is either left rotation followed by right one or right rotation followed by left one. Please consider these two links to know about rotations in details: 1. https://courses.cs.washington.edu/courses/cse373/06sp/handouts/lecture12.pdf 2. https://www.tutorialspoint.com/data_structures_algorithms/pdf/avl_tree_algorithm.pdf

Note: You are encouraged to search the internet for more details about AVL Trees or for an implementation of AVL Trees using Python. Please, try to understand the information you read on the internet and don't just copy it without exerting required effort.

[ ]:
```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
        self.height = 1
```

[ ]:
```python
class AVLTree:
    def getHeight(self, root):
        """
        Return the height of root node
        """

        if not root: return 0
        return root.height

    def getBalance(self, root):
        """
        Check for root's balance
        """

        if not root: return 0
        return self.getHeight(root.left) - self.getHeight(root.right)

    def getMinValueNode(self, root):
        """
        Get the node with lowest value (i.e., far left node)
        """

        if root is None or root.left is None:
            return root
        return self.getMinValueNode(root.left)

    def insert(self, root, val):
        """
        Insert node with target value "val"
        """
```

```
        if not root:
            return TreeNode(val)
        elif val < root.val:
            root.left = self.insert(root.left, val)
        else:
            root.right = self.insert(root.right, val)

        root.height = 1 + max(self.getHeight(root.left),
                              self.getHeight(root.right))

        ################################################################
        ##                        YOUR CODE HERE                      ##
        ################################################################
        balance = self.getBalance(root)  # get balance factor, te difference␣
↪between height of left and right subtrees
        # if +ve means tree is heavier on the left
        # if -ve means tree is heavier on the right
        #terminology reminders
        # left heave means left subtree of root height is has over 1 of␣
↪difference compared to right subtree of root or balance> 1
        # left heave means right subtree of root height is has over 1 of␣
↪difference compared to left subtree of root or balance> -1
        # single rotations cases
        if balance > 1 and val < root.left.val:  # means the tree is left heavy␣
↪and we need a right rotation
            return self.rotateRight(root)
        elif balance < -1 and val > root.right.val: # means the tree is right␣
↪heavy and we need a left rotation
            return self.rotateLeft(root)
        # double rotations cases
        elif balance >1 and val>root.right.val: # means tree is left heavy but␣
↪a node is inserted in the right subtree of a left subtree
            # so a left-right rotation must take place
            root.left=self.rotateLeft(root.left)# perform a left rotation on␣
↪the right subtree which exist in a left subtree of root
            return self.rotateRight(root) #perform normal right rotation
        elif balance < -1 and val <root.right.val: # means tree is right heavy␣
↪but a node is inserted in the left subtree of a right subtree
            root.right=self.rotateRight(root.right)
            return self.rotateLeft(root)

        return root

    def delete(self, root, val):
        """
        Delete a node with target value "val"
```

```
        """
        if not root:
            return root
        elif val < root.val:
            root.left = self.delete(root.left, val)
        elif val > root.val:
            root.right = self.delete(root.right, val)
        else:
            if root.left is None:
                temp, root = root.right, None
                return temp
            elif root.right is None:
                temp, root = root.left, None
                return temp
            temp = self.getMinValueNode(root.right)
            root.val = temp.val
            root.right = self.delete(root.right, temp.val)
        if root is None:
            return root

        root.height = 1 + max(self.getHeight(root.left),
                              self.getHeight(root.right))

        ##################################################################
        ##                        YOUR CODE HERE                        ##
        ##################################################################
        balance=self.getHeight(root)
        if balance > 1 and self.getBalance(root.left) >= 0:
            return self.rotateRight(root)

            # Case 2 - Right Right
        if balance < -1 and self.getBalance(root.right) <= 0:
            return self.rotateLeft(root)

            # Case 3 - Left Right
        if balance > 1 and self.getBalance(root.left) < 0:
            root.left = self.rotateLeft(root.left)
            return self.rotateRight(root)

            # Case 4 - Right Left
        if balance < -1 and self.getBalance(root.right) > 0:
            root.right = self.rotateRight(root.right)
            return self.rotateLeft(root)
        return root

    def rotateLeft(self, root):
        """
```

```
        Left rotate the root tree
        """
        x1=root.right
        x2=x1.left
        x1.left=root
        root.right=x2
        x1.height=1+max(self.getHeight(x1.left),self.getHeight(x1.right))
        root.height=1+max(self.getHeight(x2.left),self.getHeight(x2.right))
        return x1
        ################################################################
        ##                        YOUR CODE HERE                      ##
        ################################################################

    def rotateRight(self, root):
        """
        Right rotate the root tree
        """
        x1=root.left
        x3=x1.right
        x1.right=root
        root.left=x3
        root.height=1+max(self.getHeight(root.left),self.getHeight(root.right))
        x1.height=1+max(self.getHeight(x1.left),self.getHeight(x1.right))
        return x1
        ################################################################
        ##                        YOUR CODE HERE                      ##
        ################################################################
```

0.5 Breadth First Traversal of AVL Tree using Circular Deque

One way to make sure that your of implemenation of the AVL Tree is correct, is to traverse your tree level by level to check for the balancing property of the tree by your naked eye. Using your Circular Deque data structure above, perform level order traversal for your AVL Tree.

[ ]:
```
def printLevelOrder(self, root):
    """
    Given root node, print the root tree level by level
    """
    if root == None:  # Empty Tree
        print("Tree is empty")
        return
    cq = CircularDeque(2**root.height)
    previous_level = 1  # Keep track of level of last printed node
    current = (root, previous_level)
    cq.insertFront(current)
    while not cq.isEmpty():
        current = cq.getFront()
```

```
        if current[0] != None:
            if previous_level < current[1]:
                print()
            print(current[0].val, end=" ")
            cq.insertRear((current[0].left, current[1] + 1))
            cq.insertRear((current[0].right, current[1] + 1))
            previous_level = current[1]
        cq.deleteFront()
    ##################################################################
    ##                        YOUR CODE HERE                        ##
    ##################################################################

myTree = AVLTree()# to implement the previous code we have to use it a class␣
 ↪method
root = None
nums = [33, 13, 52, 9, 21, 61, 8, 11]
for num in nums:
    root = myTree.insert(root, num)
myTree.printLevelOrder(root)
myTree.delete(root, 13)
myTree.printLevelOrder(root)
```