**The Treap ADT**

A **Treap** stores keys and supports the following operations:

* **insert(key)** – adds *key* to the Treap. Raise AlreadyPresentError if *key* is present
* **delete(key)** - removes *key* from the Treap. Raise a KeyError if the *key* is not present
* **height** – returns the maximum distance from the root to a leaf
* **\_\_len\_\_** - returns the number of nodes in the Treap
* **\_\_contains\_\_(key)** – returns true if *key* is in the Treap

These operations may appear to be very similar to those supported by a Binary Search Tree (BST). This is because a Treap is a type of binary search tree that also has a heap property where every node is given a random priority when inserted. This allows the Treap to remain relatively balanced.

class TreapNode:

def \_\_init\_\_(self, value, left=None, right=None):

self.\_value = value

self.\_num = random()

self.\_left = left

self.\_right = right

self.\_height = 0

def \_\_str\_\_(self):

return str(self.\_value)

def get(self, key):

if self.\_value > key:

if self.\_left:

return self.\_left.get(key)

return False

if self.\_value < key:

if self.\_right:

return self.\_right.get(key)

return False

else:

return True

def updateheight(self):

if self.\_left is None and self.\_right is None:

self.\_height = 0

return self.\_height

if self.\_left is None:

self.\_height = 1 + self.\_right.updateheight()

return self.\_height

if self.\_right is None:

self.\_height = 1 + self.\_left.updateheight()

return self.\_height

self.\_height = 1 + max(self.\_left.updateheight(), self.\_right.updateheight())

return self.\_height

def rotateleft(self):

A = self

C = A.\_right

F = C.\_left

C.\_left = A

A.\_right = F

return C

def rotateright(self):

A = self

B = A.\_left

E = B.\_right

A.\_left = E

B.\_right = A

return B

def insert(self, value, node = None):

if self.\_value == value:

raise AlreadyPresentError

if self.\_value > value:

if self.\_left:

self.\_left = self.\_left.insert(value, node)

else:

self.\_left = node

if self.\_num < self.\_left.\_num:

return self.rotateright()

else:

return self

elif self.\_value < value:

if self.\_right:

self.\_right = self.\_right.insert(value, node)

else:

self.\_right = node

if self.\_num < self.\_right.\_num:

return self.rotateleft()

else:

return self

def display(self, offset=0):

offset += 2

if self.\_left:

self.\_left.display(offset)

print(' ' \* offset, end='')

print(self,end=' : ')

print(self.\_num)

if self.\_right:

self.\_right.display(offset)

def delete(self, key):

if self.\_value > key:

if self.\_left:

self.\_left = self.\_left.delete(key)

return self

raise KeyError

elif self.\_value < key:

if self.\_right:

self.\_right = self.\_right.delete(key)

return self

raise KeyError

elif self.\_value == key:

self = self.deletethisnode()

return self

def deletethisnode(self):

if self.\_left is None:

self = self.\_right

return self

elif self.\_right is None:

self = self.\_left

return self

elif self.\_left.\_num > self.\_right.\_num:

self = self.rotateright()

self.\_right = self.\_right.deletethisnode()

return self

else:

self = self.rotateleft()

self.\_left = self.\_left.deletethisnode()

return self

The TreapNode class is the heart of this Treap implementation. The Treap class below handles all the operations, but most of them call a TreapNode function on the root of the Treap. This class is similar to a normal Node class used in a BST except that the initializer creates a priority using the random function. The functions used to rotate left and rotate right rearrange the nodes as defined by these tree rotations. The insert function uses binary search to find the location where the key should be inserted which may disrupt the heap property. On the way up the traversal, rotations are performed to fix the heap property while keeping the BST property. The delete function finds the location of the key using binary search which runs in O(log n) and then calls deletethisnode to remove it using tree rotations to get the node down to a leaf while maintaining the BST and heap properties. Finally, display and updateheight are two functions that I implemented in order to understand the Treap. Display prints an in-order traversal of the tree with the current node as the root, so when this is called on the root of the tree, it displays a complete in-order traversal. The offset is used in order to indent nodes at different level by adding to the offset with every recursive call. Update height is used to examine the height of the tree and it can be seen that it is the max distance from the current node to a leaf because a leaf has a height of 0 and every other node’s height is defined as 1 more than the height of the node’s child that has the maximum height.

class Treap:

def \_\_init\_\_(self):

self.\_root = None

self.\_length = 0

def insert(self, key):

node = TreapNode(key)

if self.\_root:

self.\_root = self.\_root.insert(key, node)

else:

self.\_root = node

self.\_length += 1

def \_\_contains\_\_(self, key):

return self.\_root.get(key)

def display(self):

if len(self) == 0:

return None

self.\_root.updateheight()

self.\_root.display()

def delete(self, key):

if self.\_root:

self.\_root = self.\_root.delete(key)

self.\_length -= 1

def \_\_len\_\_(self):

return self.\_length

def height(self):

if self.\_root:

self.\_root.updateheight()

return self.\_root.\_height

The Treap class in itself seems kind of like a wrapper for the TreapNode class, which is true in a way, but they represent different objects. The Treap class represents the whole tree by only storing the root and the length of the tree which is given by the number of nodes in the tree. Since the TreapNode class has so many operations, it is simple to just store the root and then call the associated function on the root. However, if there is no root, then insert, delete, display, and height do nothing.