**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’s height to remain O(log n) with high probability and the analysis of this is shown later.

The Treap class below seems to be a wrapper for the TreapNode class, 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. This is a simple implementation of the Treap class where most of the operations are delegated to the TreapNode class. However, if there is no root, then delete, display, and height do nothing. From the Treap class, we can only do an analysis of the length operation which runs in constant time because it is stored and updated within the Treap class. The other operations; insert, contains, display, delete, and height are dependent on their TreapNode operations counterpart.

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):

if self.\_root:

return self.\_root.get(key)

else:

return False

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 TreapNode class below is the heart of this Treap implementation. 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 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. The rotate functions, rotate left and rotate right, rearrange the nodes in a certain way that does not disrupt the BST property. As long as these tree rotations are used appropriately as shown in deletethisnode and insert, then the heap property is not violated either.

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 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

Analysis:

We have already learned that \_\_len\_\_ runs in constant time in all cases. The helper functions rotate right, rotate left, and \_\_str\_\_ run in constant time as well because there are no loops or recursion. Next, it can be seen that updateheight and display have recursion calls to both children if they exist. Therefore, theses operations run in O(n) time in all cases because it must traverse every node and there is a constant time call made at each node.

The trickier, yet more essential operations to analyze are insert, get, and delete because they depend upon the shape of the Treap. These operations make at most one recursive call and only consist of constant time operations. This means that the running time is proportional to the height of the Treap. Therefore, the worst case for these operations is O(n) if the Treap is very unbalanced and proving that the average case running time is O(log n) requires proving that the average height of the tree is O(log n).

The average height of the tree can also be found by calculating the average expected node depth. This can be given by the following formula:

Average Expected Depth =

This formula comes about by understanding that the expected value of the depth of node xi can be given by the summation of the probabilities that xm is an ancestor of xi. The probability that xm is an ancestor of xi is the same as the probability that the priority assigned to node xm is larger than other priorities associated with the nodes m through i. This probability can be simplified because it is equally likely that any node xi through xm, so any of , will have the largest priority since the priorities are assigned randomly. This formula can be written as:

Finally, by simplifying the double summation in the first equation, we are left with the equation for the average height of the Treap where C is a constant:

Average Height = Average Expected Depth =

From this we can deduce that the average height of the Treap is O(log n) which means that the average case running time for insert, delete, and contains are also O(log n).

This result is independent of the order that the data elements are inserted. In fact, a regular BST would create the same tree as the Treap if the elements are inserted in order of decreasing priority. However, this limits the BST to the fact that all of the data must be present at the time of first insertion. It is much more reasonable to use a Treap in order to provide liberty within the data insertion process.

These results can also be found experimentally by comparing the height of a Treap to the height of a BST with elements inserted in different orders. When the elements are inserted in random order, the heights of the trees are comparable. However, when inserted in sorted order which is common, the height of the BST is proportional to the number of elements while the height of the Treap is proportional to the log of the height of the elements and still comparable to the height when the elements were inserted randomly. These results are shown below.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Random Height Test

Elements were inserted in random order

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Complete Tree would have height of 1

Height of BST is: 1

Height of Treap is: 1

Complete Tree would have height of 3

Height of BST is: 5

Height of Treap is: 7

Complete Tree would have height of 5

Height of BST is: 12

Height of Treap is: 9

Complete Tree would have height of 7

Height of BST is: 16

Height of Treap is: 16

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Ordered Height Test

Elements were inserted in sorted order

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Complete Tree would have height of 1

Height of BST is: 2

Height of Treap is: 1

Complete Tree would have height of 3

Height of BST is: 14

Height of Treap is: 5

Complete Tree would have height of 5

Height of BST is: 62

Height of Treap is: 11

Complete Tree would have height of 7

Height of BST is: 254

Height of Treap is: 14

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*