*Design document*

Binary Search Tree (BST)

About BST:

A Binary search tree is a rooted binary tree data structure where key of each internal node will be greater than all the keys in it’s left subtree and less than that of in right subtree.

Custom BST that we implemented in our problem statement compares Priority of Strings.

high priority given to Special character(! (least) @ # $ ^(highest)) only use these 5 special characters in the same order of priority as mentioned, next priority to numeric values ( 0 (least) to 9 (highest) ) and last priority to alphabets (small letters (least), capital letters (highest))

^ > $ > # > @ > ! > 9 > 8 > 7 > 6 > 5 > 4 > 3 > 2 > 1 > 0 > Z > Y > X > W > V > U > T > S > R > Q > P > O > N > M > L > K > J > I > H > G > F > E > D > C > B > A > z > y > x > w > v > u > t > s > r > q > p > o > n > m > l > k > j > i > h > g > f > e > d > c > b > a

If the same/duplicate word is given as input then add a counter value of that node and increment it

using above constraints we need to answer the following questions,

1. Figure out how many characters in each input value of all the nodes in the tree?

2. Identify how many nodes in the tree are with the same string count and mention those words under the relevant count list.

3. List out all the given input words based on the order of priority as mentioned in the problem statement from highest to least/least to highest.

4. Find out the largest string count and the duplicate string after comparing all the nodes in the tree and those nodes position/level in the binary search tree.

5. Specify all the node input values in the preorder traversal of BST.

Implementation:

> created a class named “Node” having attributes “data” , “count” , “left” , “right” and method “less\_priority\_than”

data : stores string word

count : stores number of occurrences of duplicate words

left : maintains reference to left node/subtree

right : maintains reference to right node/subtree

length: counts the length of Word while creating Node object , O(w) where w is length of word

less\_priority\_than : compares character by character priority between self.data and word to be compared by passing it as a parameter to this function.

If less priority returns 0 else 1

if shortest word is part of other and has less or same priority for common part then we will check priority based on length.

**less\_priority\_than** method has time complexity as **O(L)**, where L is length of largest word.

> Inserting into Binary Search Tree

method “insert”

gets priority using less\_priority\_than method if more priority then insert into right subtree recursively else insert into left subtree recursively.

If the word is already (checked by method less\_priority\_than returning -1) in BST then it increments the ‘count’ attribute of Node

**Insert** has time complexity is **O(h)**, where ‘h’ is height of the BST.

But we count the length of word which is O(w) where w is length of word

Overall time complexity is O(h x w)

> Inorder Traversal:

method “inorder\_traversal\_des”

here we visit each node in BST I.e by processing right subtree first then print the current node data finally processing left subtree to print word from highest priority to lowest priority.

since we are anyhow visiting each and every node once, doing two additional checks

Duplicates

Largest Word

Duplicates, are checked by using the ‘count’ which was populated while inserting into BST itself. If count>1 then mark it as duplicate.

Largest Word, is find by comparing length attribute of node object which was created during object creation and mark it as largest.

Display Duplicates and Largest Word in next step respectively.

Overall time complexity is O(h)

> Preorder Traversal

here we visit each node in BST by printing out current node data first then processing left subtree and finally processing right subtree

Overall time complexity is O(h)