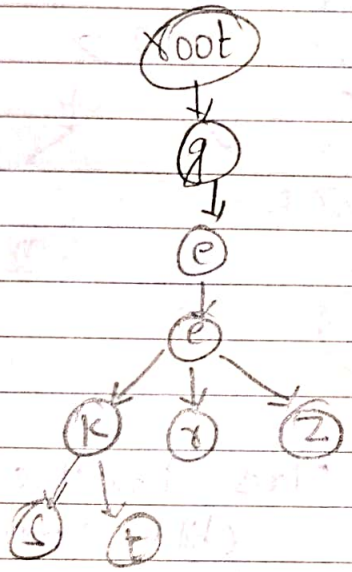
Trie

⇒ The trie data structure is an efficient information retrieval DS. The trie data structure is used to efficiently search for a particular string key among a list of such keys.
Using trie, the lookup operation can be performed in TC = O(key-length).

⇒ A trie is represented as a tree where each node contain 26 pointers which is equal to the number of character in the English alphabet.
In Trie basically, the common prefix of all string are represented as a common path in the tree.



Efficient for the following operation on words in dictionary.
ⓐ Search of words.
ⓑ Insert of words.
ⓒ Delete of words.
ⓓ prefix search of words.
ⓔ lexigraphical ordering of words.

4 different strings :-
['geeks', 'geek', 'geer', 'gee2']

Comparision with hashing

	Trie	hashing
Search	O(word-len) in worst case	O(word-len) on average
Insert	"	"
Delete	"	"
Prefix search	O(prefix-len + output-len)	Not supported
Lexographic ordering	O(output-len)	Not supported

DECEMBER
03
'24
TUESDAY
49th Week
338-028

Trie Representation

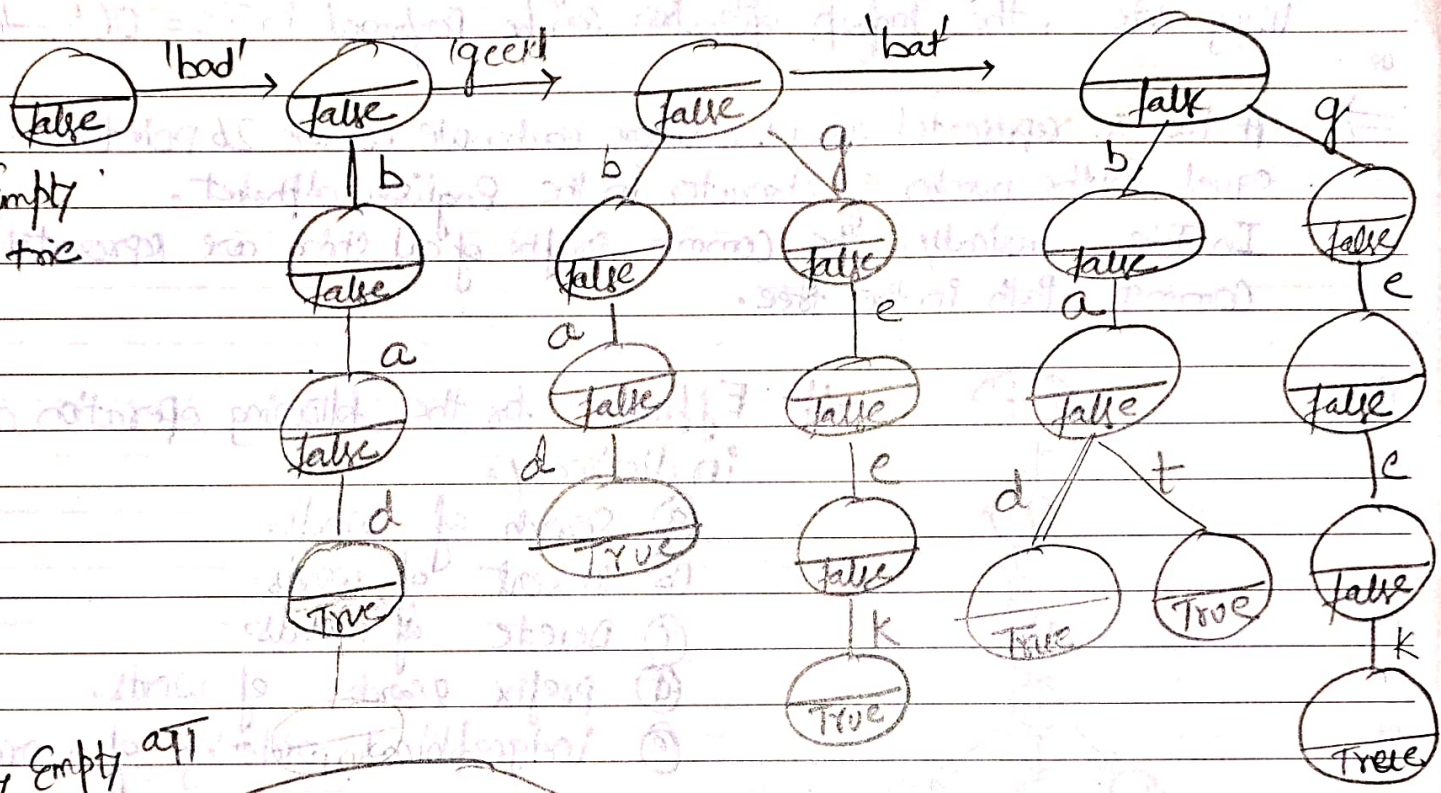
S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

DEC 2024

→ Binary tree is empty when root is None.

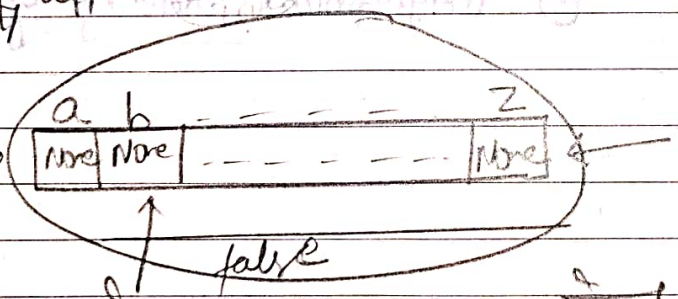
→ Trie is empty when root is containing all the 26 children None.

Insert



Empty trie

Initially Empty आती



वहाँ पर reference store करेंगे।

Class TrieNode:
Child = [None] * 26
isEndOfWord = False

Insertion - inserting a key of Trie is a simple approach. Every character of input key is inserted as an individual Trie node. Note that the children are an array of pointers to next level trie node. The key character acts as an index into the array of children. If the input key is new or an extension of the existing key, we need to construct non-existing nodes of key, and mark end of word for the last node. If the input key is a prefix of the existing key in Trie, we simply mark the last node of the key as the end of word. The key length determine Trie depth.

```
def insert(node, key):
    for ch in key:
        i = ord(ch) - ord('a')                    TC = O(len(key))
        if node.child[i] is None:
            node.child[i] = new TrieNode()
        node = node.child[i]
    node.isEndOfWord = True
```

Search — Searching of a key is similar to the insert operation, however we only compare the characters and move down. The search can terminate due to the end of a string or lack of key in the trie. In the former case, if the isEndOfWord field of last node is true, then the key exist in the trie. In the second case, the search terminates w/o examining all the character of the key, since the key is not present in the.

```
def search(node, key):
    for ch in key:
        i = ord(ch) - ord('a')                    TC = O(len(key))
        if node.child[i] is None:
            return False
        node = node.child[i]
    return node.isEndOfWord
```

Delete
- We write a Recursive function.
- We first traverse to the last node we mark it not end of a word.
- If this node is empty (contain 0 children) we remove it.
- If parent of this node had only one child and not end of a word, we remove parent as well.
- We repeat the above step for parent of parent and so on.

DECEMBER
05 '24
49th Week
340-026
THURSDAY

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

DEC 2024

Implementation

```
def delNode(root, key, i=0):
    if root == None:
        return None
    if i == len(key):
        if root.isEndOfWord:
            root.isEndOfWord = False
        if isEmpty(root):
            root = None
        return root
    index = ord(key[i]) - ord('a')
    root.child[index] = remove(root.child[index], key, i+1)
    if isEmpty(root) and root.isEndOfWord == False:
        root = None
    return root
```

```
def isEmpty(root):
    for n in root.child:
        if n != None:
            return False
    return True
```

TC = O(n)

Count distinct rows in a Binary matrix.

i/p: mat = [[1,0,1],
[1,1,1],
[1,0,0],
[1,1,1]]

o/p = 2

appeared before

i/p: mat = [[1,0,0]
[1,0,0]]

o/p: 1.

Naive Approach

① Initialize: res = 0

② Traverse through all rows

(a) If the current row does not match any of the previous rows:

res = res + 1

③ return res

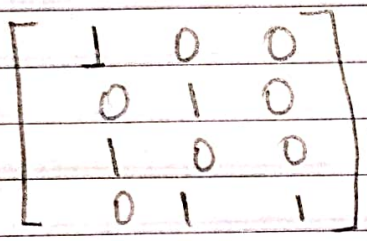
(column)

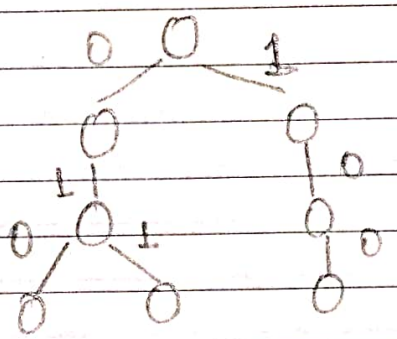
O(R²×C)

Row

Efficient Approach

(1) Create an Empty trie
(2) Initialize : res = 0
(3) Do the following for every row:
  (a) Consider the row as a word and insert into the trie.
  (b) If the row was not already there res = res + 1.
  (c) If row was already there don't do anything.
(4) return res





TC = O(R * C)

```
Class TrieNode:
    def __init__(self):
        self.child = [None]*2

def insert(node, row):
    isNew = false
    for i in row:
        if not node.child[i]:
            node.child[i] = TrieNode()
            isNew = True
        node = Node.child[i]
    return isNew
```

```
def countDist(mat):
    res = 0
    root = TrieNode()
    for i in range(len(mat)):
        if (insert(mat[i])):
            res = res + 1
    return res.
```