Tries

~~~~~~~

Agenda

~~~~~~~

1. Need of TRIE : Spell Checker
2. Introduction to TRIE
3. TRIE function
   - * Insert
   - * Search
   - * Deletion
4. Shortest Unique Prefix

Hello Everyone
Very Special Good Evening
to all of you 😊😊😊
We will start session
from 9:06 PM

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Need of TRIE : Spell Checker

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

type: playgroind
loyal
friend
lyfe

task: Given a spelling, check if spelling is correct or not?

* To implement this problem in efficient way (we have to use TRIE)
  # Hierarchal Data structure
  # Prefix-Tree
  # TRIE
  → It is used in information retrieval

Rough logic:

* Maintain a Dictionary
* check if that word available in dictionary or not.

Words: * Apple ✓
* Appeal x Not present in dictionary.

Dictionary:
* Apple
* Anatomy
* Life
* Daily

Search for Word

Given an array of Strings and a word, check if that word is present in array or not.

arr: [drink , draw , drone , cat , cattle , Car , mango , man]

Word	Ans
Cat	true
Cate	false
draw	true

#Similar to spell checker

#Idea1: Add all strings in Hashset and check if word is present in Hashset or not.

Important function of Hashset:
1. HashCode → Helps to uniquely identify key
2. equals → Compare input with key

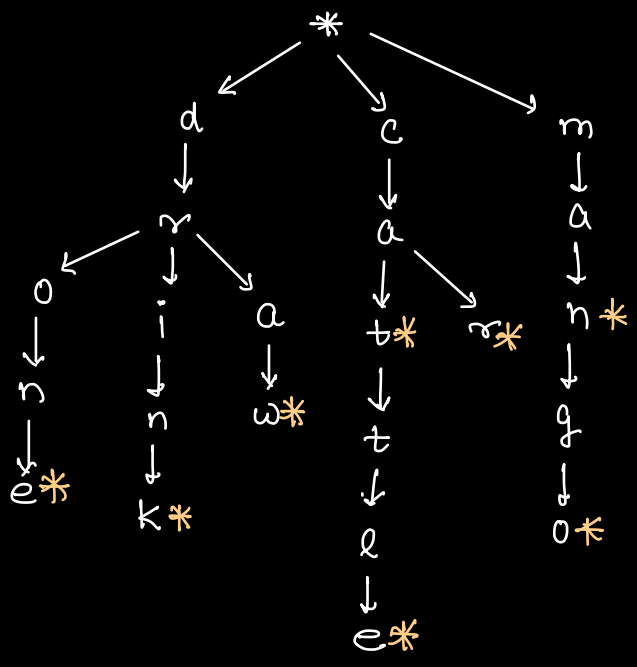
* l → average length of string
* N → words are there

T.C: O(N * l)
S.C: O(N * l)
} Preparation of Hashset i.e. dictionary

Introduction to TRIE

We can improvise it using TRIE data structure

arr: [drink , draw , drone , cat , cattle , Car , mango , man]



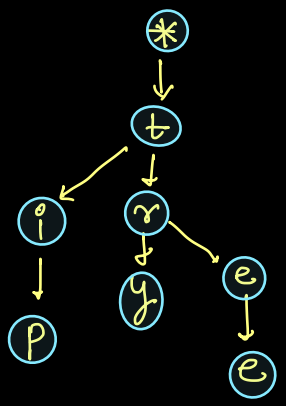
To find:
* Cat → true
* draw → true
* cate → false
* mang → false

*] marker: It represent end of word ⇒ eow

TRIE function

* Insert
* Search
* Deletion

Example: try tree tip

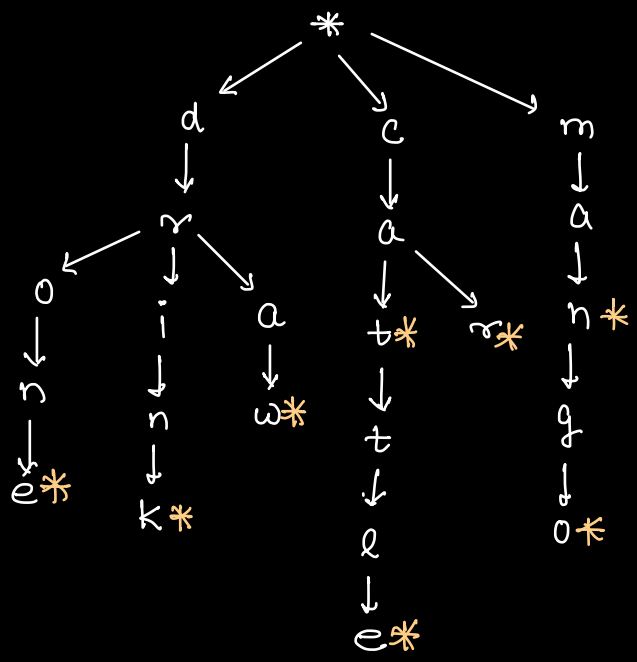


Node = 8

Insertion:

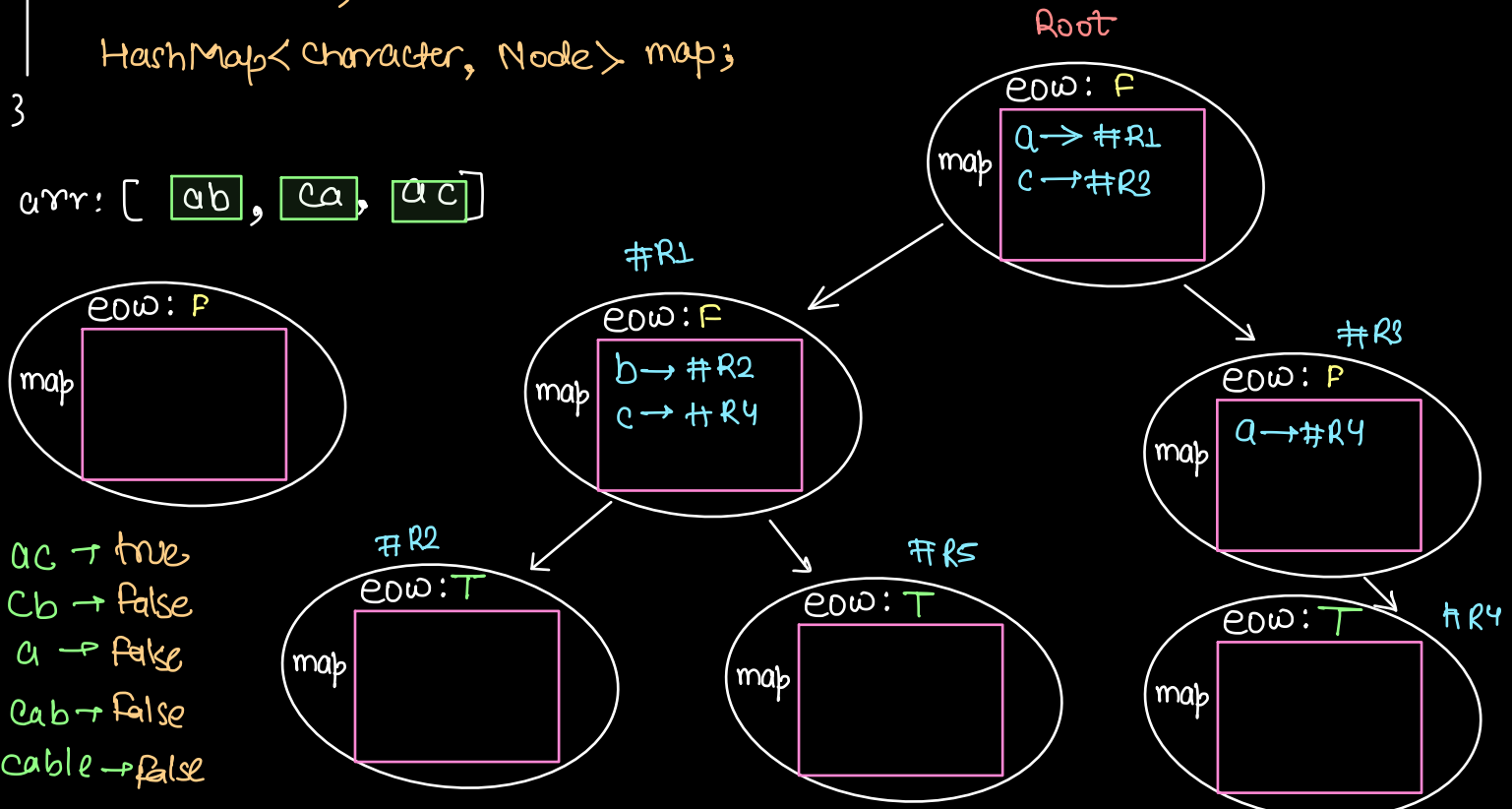
```
Class Node {
    boolean eow;
    HashMap<Character, Node> map;
}
```

arr: [drink, draw, drone, cat, cattle, car, mango, man]

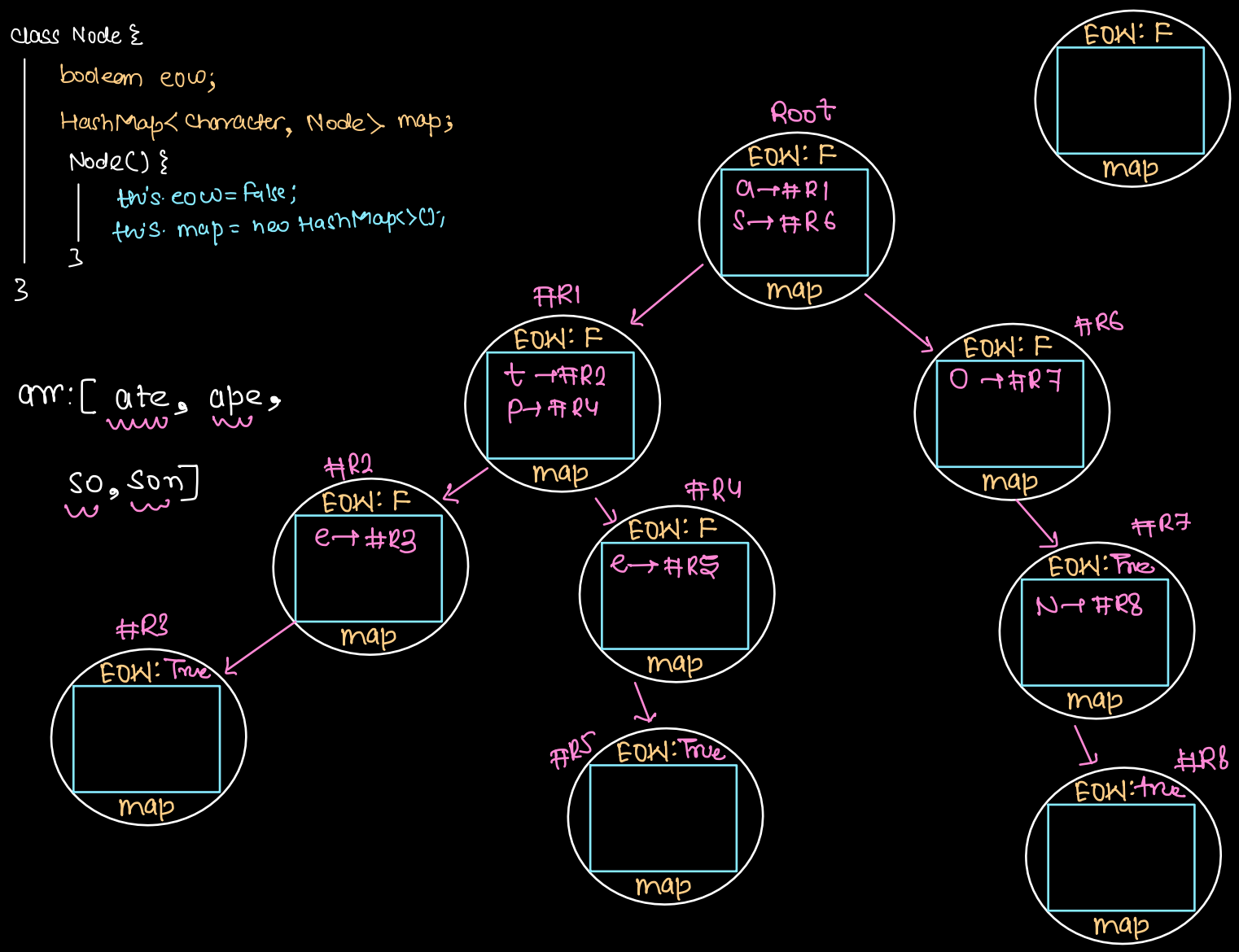



```
Class Node {
    boolean eow;
    HashMap<Character, Node> map;
}
```

arr: [ab, ca, ac]



```
class Node {
    boolean eow;
    HashMap<Character, Node> map;
    Node() {
        this.eow = false;
        this.map = new HashMap<>();
    }
}
```



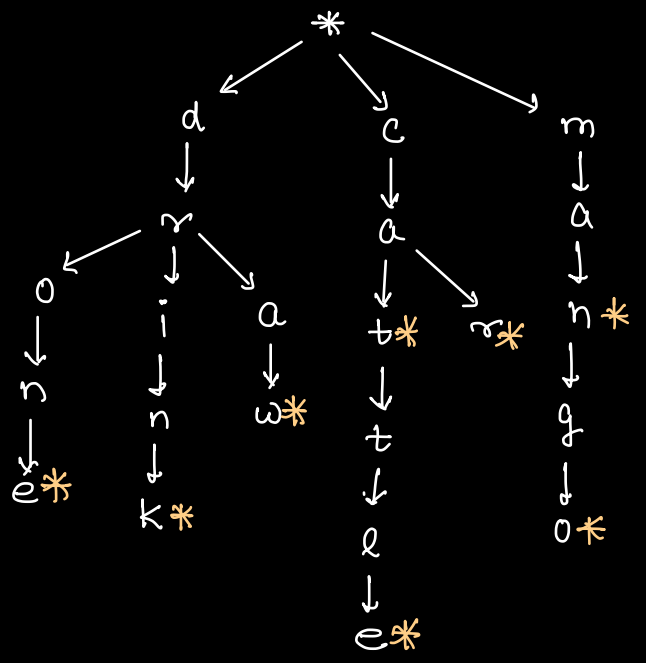

```
void insert(Node root, String word) {
    Node temp = root;
    for(int i=0; i< word.length(); i++) {
        char ch = word.charAt(i);
        if(temp.map.containsKey(ch)){
            // if available, move temp to that address
            temp = temp.map.get(ch);
        } else {
            // if not available, create a new Node
            store it in map of current node then
            move on that created node.
            Node nn = new Node();
            temp.map.put(ch, nn);
            temp = nn;
        }
    }
    temp.eow = true;
}
```

T.C: O(l)
S.C: O(l)
→ l = avg length

Searching

→ Assumption: Trie is prepared

word	Response
mang →	false
can →	false
man →	true
drift →	false



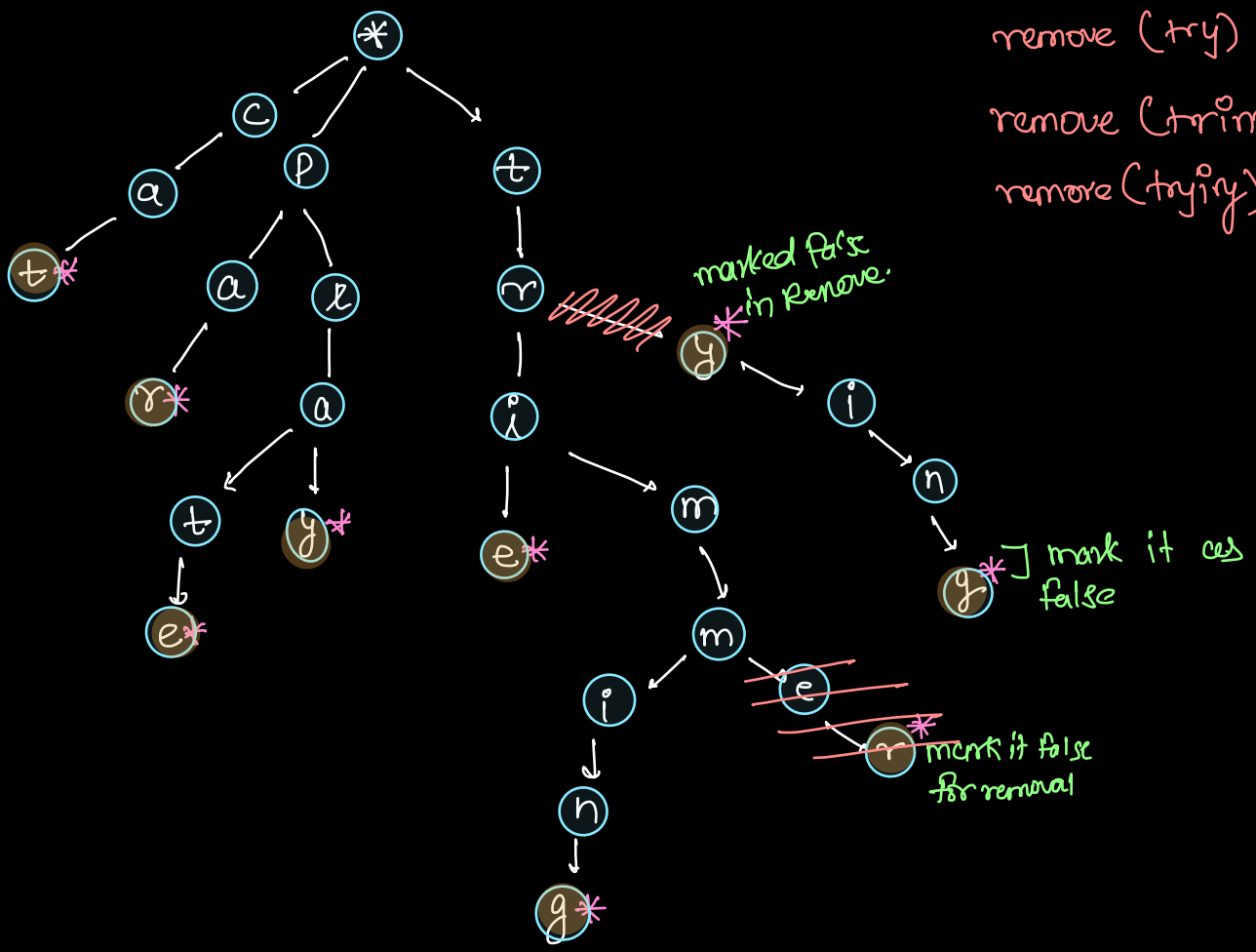

```
boolean Search(Node Root, String word) {
    Node temp = Root;
    for(int i=0; i< word.length(); i++){
        char ch= word.charAt(i);
        if(temp.map.containsKey(ch)) {
            temp= temp.map.get(ch);
        } else {
            return false;
        }
    }
    return temp.eow;
}
```

For Searching-

S.C: O(1)

T.C: O(n*l)

#deletion:



NOTE: if word which we want to remove is prefix substring of another word the we can't remove any no. of node.

⟶ just make eow = false.

Condition for Node which we can't Remove:

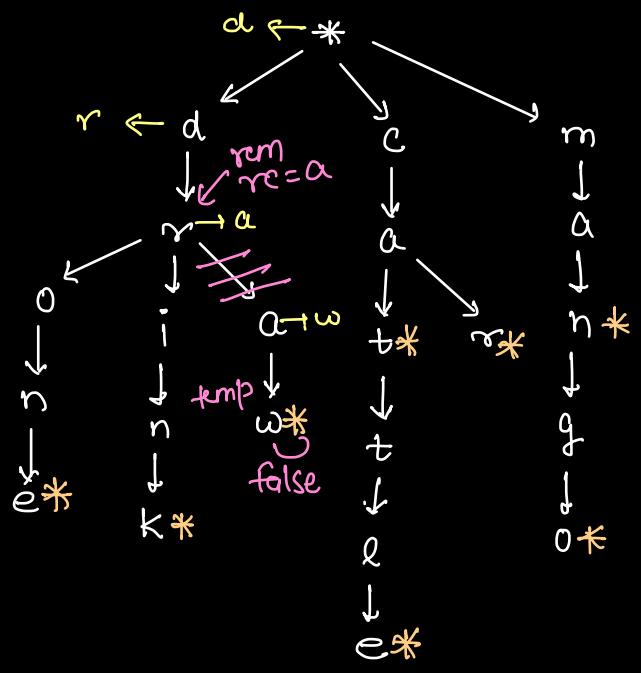
1. If no. of children[map.size()] is more than one then we can't remove it.
2. If a node is marked as true, we can't remove it because it can be ending point of another word.

draw → Remove

rem → Hold the node Address from where we can Remove the connection

rc → removal character

temp → Help us to iterate on trie.



rem = Root

temp = root

rc = d

```
void deletion(Node root, String word) {
    Node temp = root, rem = root;
    char rc = word.charAt(0);
    for(int i=0; i< word.length(); i++) {
        char ch = word.charAt(i);
        // count no. of children for that character.
        int count = temp.map.size();
        if(count > 1 || temp.eow == true) {
            rem = temp;
            rc = ch;
        }
        temp = temp.map.get(ch);
    }
    temp.eow = false;
    if(temp.map.size() == 0) {
        // if no further links available after temp,
        then de-link from removal node
        rem.map.remove(ch);
    }
}
```

T.C: O(L)

S.C: O(1)

Shortest Unique Prefix

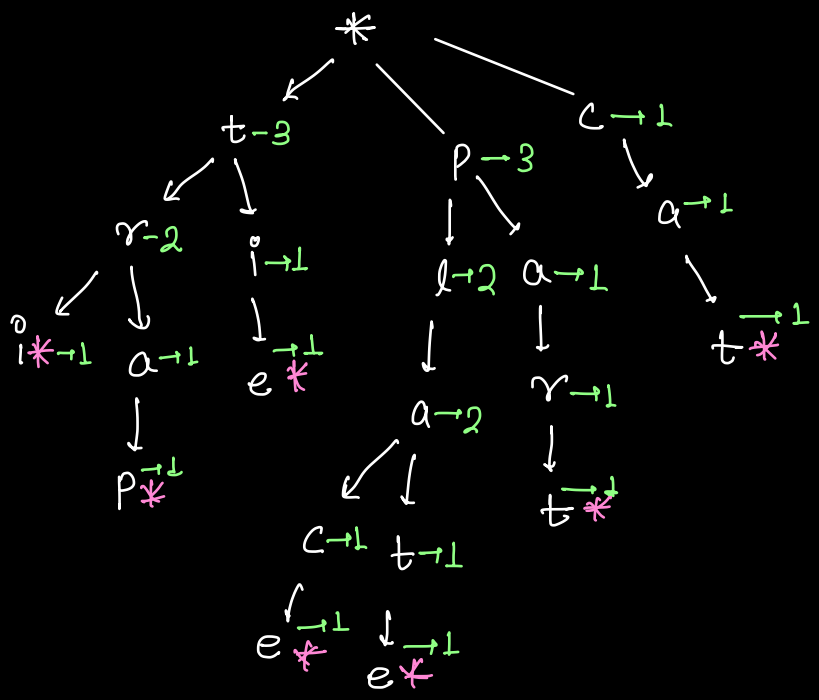
Find the shortest unique prefix substring to represent each word.

Note: Assume that no word is prefix of another word.

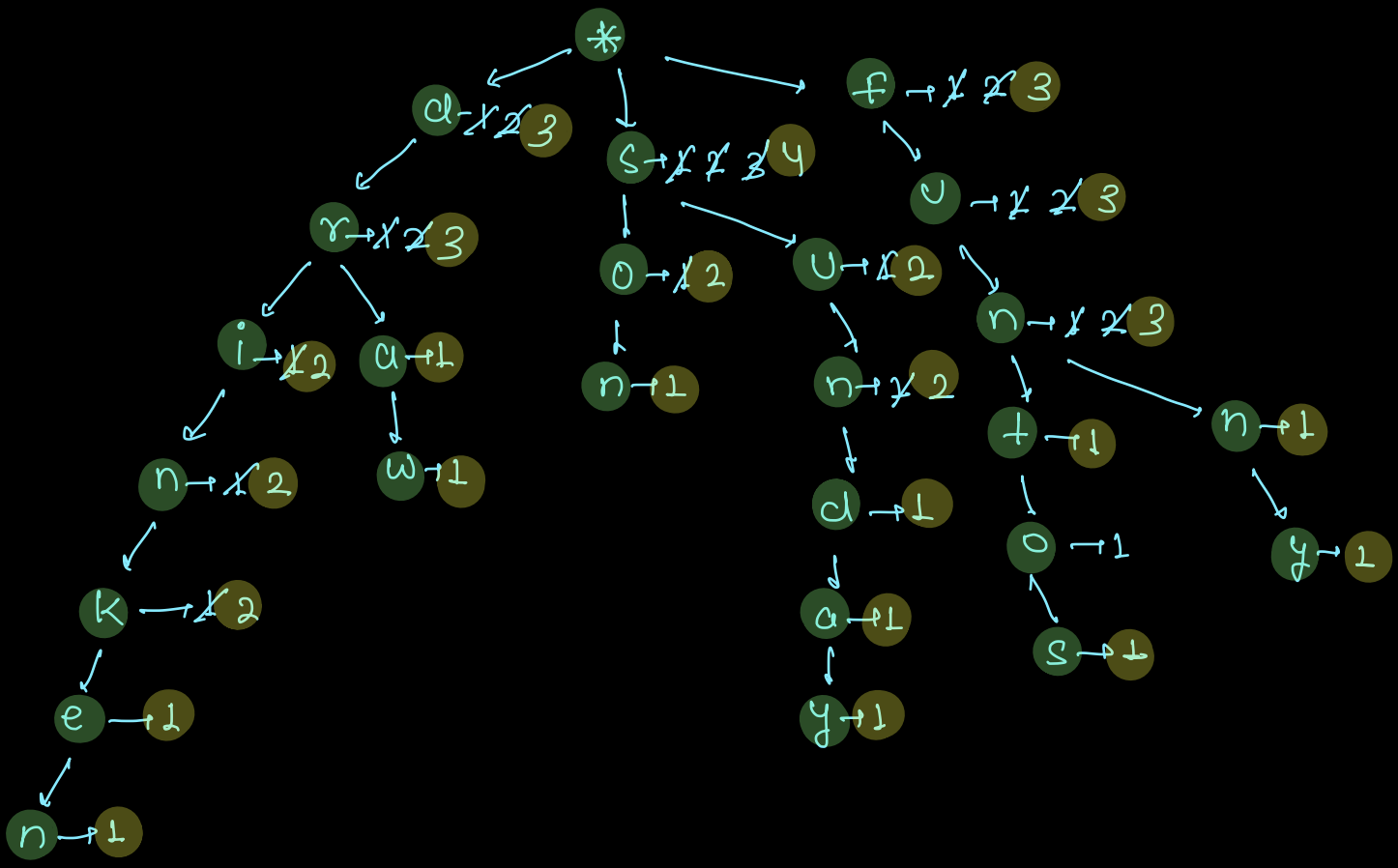
> In other words, the representation always possible

words: [tri, trap, plate, cat, part, place, tie]

shortest unique prefix → tri tra plat c pa plac ti



dictionary: [drink, draw, drinker, so, son]
fun, funtos, funny, son, sunday



Shortest prefix for sunday → SUND

How many strings have "fun" is available as substring ??

Implementation:

```
Node {
    int pc=0;  // path count
    map;
    Node(){
        pc=0;
        map: new HM<>();
    }
}
```

NOTE: End of word flag not require in this problem.

Code change in insertion;

```
void Insert(Node root, String word) {
    Node temp = root;
    for(int i=0; i< word.length(); i++) {
        char ch = word.charAt(i);
        if(temp.map.containsKey(ch)){
            temp= temp.map.get(ch);
            temp.pc++;
        } else {
            // if not available, create a new Node
            store it in map of current node then
            move on that created node.
            Node nn = new Node();
            temp.map.put(ch, nn);
            temp = nn;
            temp.pc++;
        }
    }
}
```

solve main problem → iterate on newly created trie.
while moving make answer, once path count = 1
return created answer.

dictionary → [m, ma, man, mang]

NOTE: map.size() will not work.