# Trie

15 September 2023 11:36 AM

# Implement TRIE | INSERT | SEARCH | STARTSWITH

Trie is used when we need to insert[word], search[word], startwith[word] means any words whose prefix is word

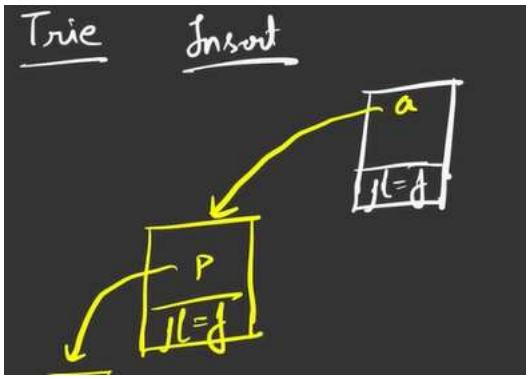
Structure of Trie

Trie is a sturct or class which has an array of 26 size of type className, bool flag
In inside array 0 means a,1, means b, 2 means c……, 25 means z

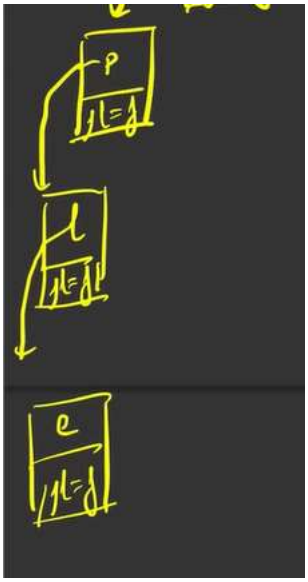

## Insert in Trie

Let say we take word apple
We see a letter if its not present we insert it in trie in a way that we create a trie node of arr[26] and bool flag and insert arr[1] = a and we create a reference trie of 'a'

Now we see next letter as 'p'
We check 'p' in trie and store It in reference of 'a' Trie node and create a reference trie node of 'p'



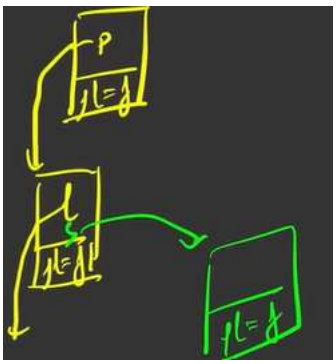

Sameway we do this for all left over characters




We insert 'e' at reference node of 'l' and we make a reference trie of 'e' also
Initially flag = false


We are at 'e' means word is completed so we mark flag = true for reference trie node of 'e'
We have finished for 'apple'

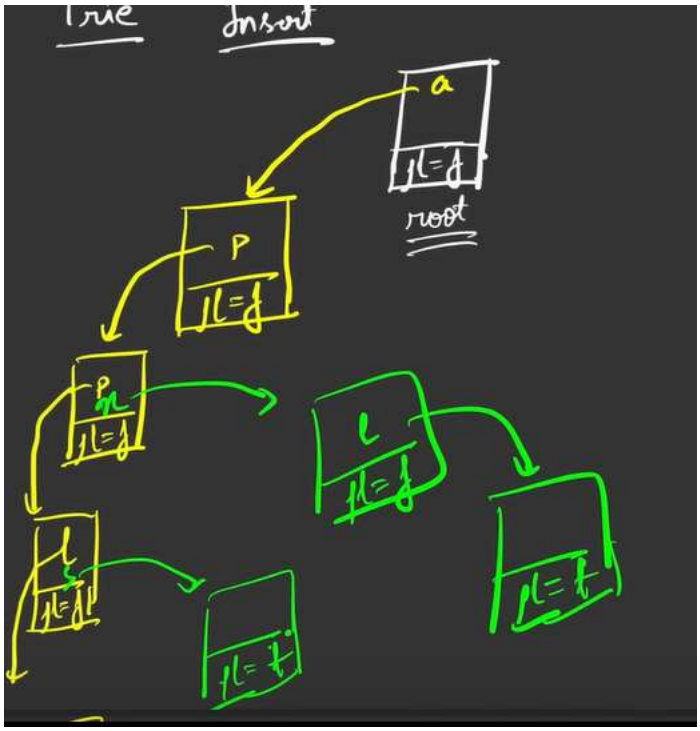
Now let's see for 'apps'
We check in 'apple' and we find that 'app' are found but no 's' found
So we insert 's' in array of trie node below 'p' which is 'l' and make a ref trie node in refrence node of 'p' which is after 'l' node in trie with bool false



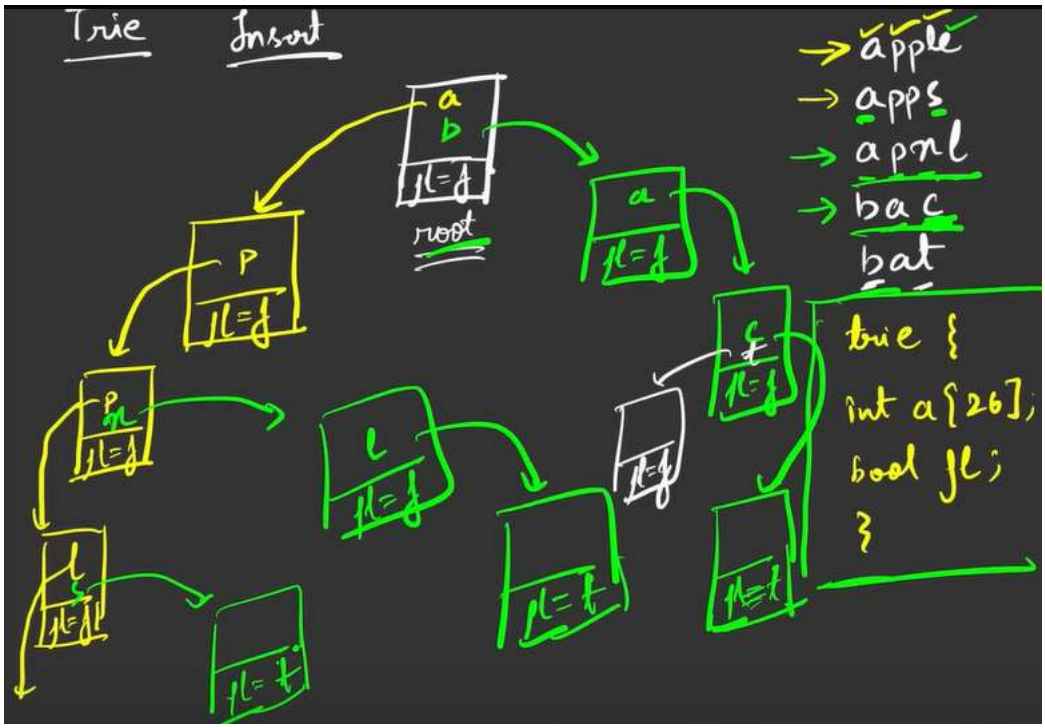

apps is complete so we mark flag = true for new reference node created below 'l'
Which shows that this is another word

Now we check for


"apxl" we see we don't have x so we create x in array of refrence node of 'p' in apple
And we need 'l' now after x so we create one more node
Word is completed so mark bool  = true which shows one more word done.




Now we go for "bac" then "bat"

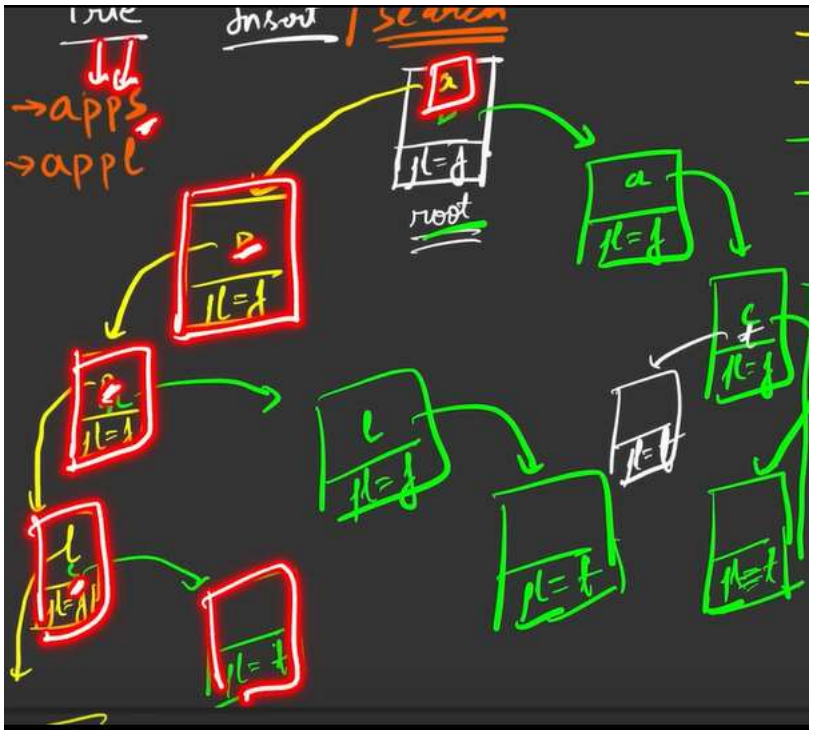



We have a root, and when we insert a element we make sure to make a reference node also, initially bool of reference node is false
When word ends, we do mark bool flag = true to mark that word is ended now.

If we have to insert a word, we insert it in reference node ka array of last matched character, like for apple and apxl
We inserted x in ref node of first p of apple i.e with second p and then we make a reference node also with initially false as we have inserted a new element.

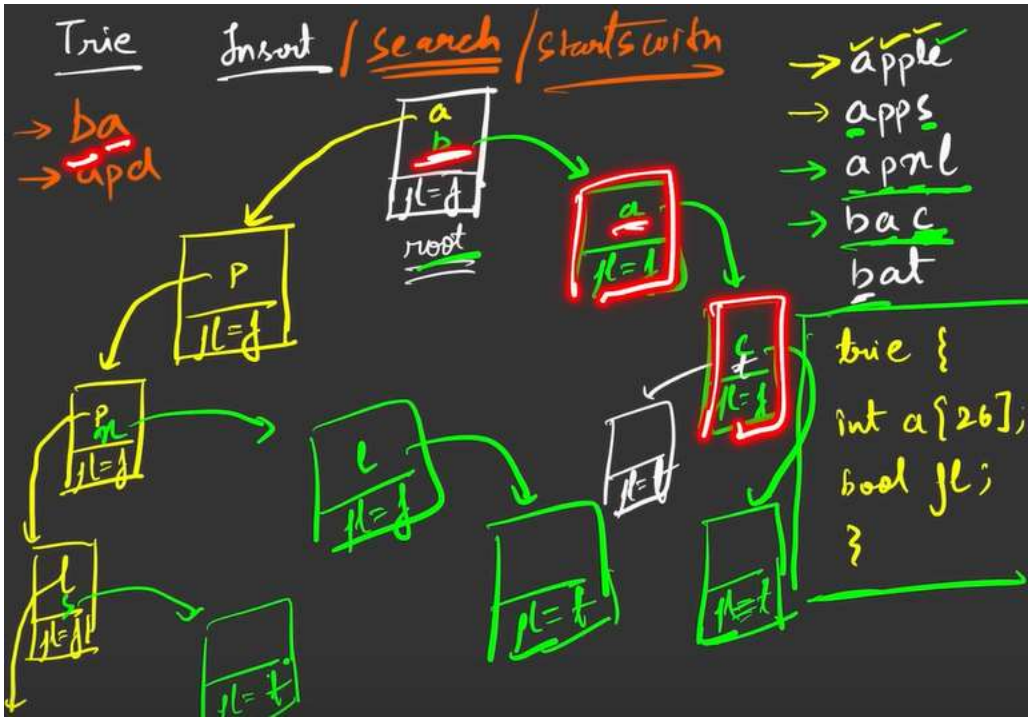
## Search in Trie

do you have "apps", we start from root and we search on and check if last ref node we are starting is true or not. If true, means yes we have found it.
If flag = false means there is not word like "apps" in our TRIE




## StartsWith

Is there any word which starts with "ba"
No, we start from root and we see there is "b" and we have "a" we move ahead
If we are standing at a trie node and its **not NULL** means there is such word with prefix "ba"




If we are encountering a NULL means there does not exits any such word

## Code

```
struct Node{
    Node *arr[26];
    bool flag = false;
    // To check if our array already has ch or not before inserting
    bool hasLetter(char ch)
    {
        // if node of that character is not NULL means that exists
        return (arr[ch - 'a'] != NULL);
    }
    // To insert a node in array for ch
    void put(char ch, Node *node)
    {
        arr[ch-'a'] = node;
    }
    // To get ref node of ch
    Node* get(char ch)
    {
        return arr[ch-'a'];
    }
    // To set the end ref node ka bool as true
    void setEnd()
    {
        flag = true;
    }
    // Check if end ref node has flag value = true means word complete
 else false, then word is not found completely
    bool isEnd()
    {
        return flag;
    }
};
class Trie {
    private:
    Node* root;
public:
    /** Initialize your data structure here. */
    Trie() {
        // Every Time constructor is called, make a new root
        root = new Node();
    }

    /** Inserts a word into the trie. */
    void insert(string word) {
        // TC: O(length of word)
        // we need to insert each letter of word in trie
        // starting from root
        Node* node = root;
        for(int i = 0;i<word.size();i++)
        {
            // before inserting check if that already exist so we make
 a function in struct Node
            if(!node->hasLetter(word[i]))
            {
                // we have used ! means if we here, it does not exist
                // let us insert it
                node->put(word[i], new Node());
```

```
            }
            // move to reference of trie node
            node = node->get(word[i]);
        }
        // loop ends means we are at the last ref node means mark the
bool = true for last ref node as word is finished
        node->setEnd();
    }

    /** Returns if the word is in the trie. */
    bool search(string word) {
        // TC: O(length of word)
        // always start with root node
        Node* node = root;
        for(int i = 0;i<word.size(); i++)
        {
            // check if char is there or not
            if(!node->hasLetter(word[i]))
            {
                // if it does not exists means
                return false;
            }
            // else give me ref node of that char
            // we are storing ref node so that we can check at the end
 if last ref node is true means complete word is found
            node = node->get(word[i]);
        }
        // now we are at end ref node so check bool
        // if true means we are at the end else not
        if(node->isEnd())
        {
            return true;
        }
        return false;
    }

    /
** Returns if there is any word in the trie that starts with the given
 prefix. */
    bool startsWith(string word) {
        // TC: O(length of word)
        // again start from root
        Node* node = root;
        for(int i = 0;i<word.size();i++)
        {
            if(!node->hasLetter(word[i]))
            {
                return false;
            }
            node = node->get(word[i]);
        }
        // if we reach here means there is something whose prefix is w
ord
        return true;
    }
};
```

# Implement Trie-2 | INSERT | countWordsEqualTo() | countWordsStartingWith()

Ninja has to implement a data structure ”TRIE” from scratch. Ninja has to complete some functions.

1) Trie(): Ninja has to initialize the object of this “TRIE” data structure.

2) insert(“WORD”): Ninja has to insert the string “WORD”  into this “TRIE” data structure.

3) countWordsEqualTo(“WORD”): Ninja has to return how many times this “WORD” is present in this “TRIE”.

4) countWordsStartingWith(“PREFIX”): Ninjas have to return how many words are there in this “TRIE” that have the string “PREFIX” as a prefix.

5) erase(“WORD”): Ninja has to delete one occurrence of the string “WORD” from the “TRIE”.

Note:

1. If erase(“WORD”) function is called then it is guaranteed that the “WORD” is present in the “TRIE”.

2. If you are going to use variables with dynamic memory allocation then you need to release the memory associated with them at the end of your solution.

**Sample Input 1:**

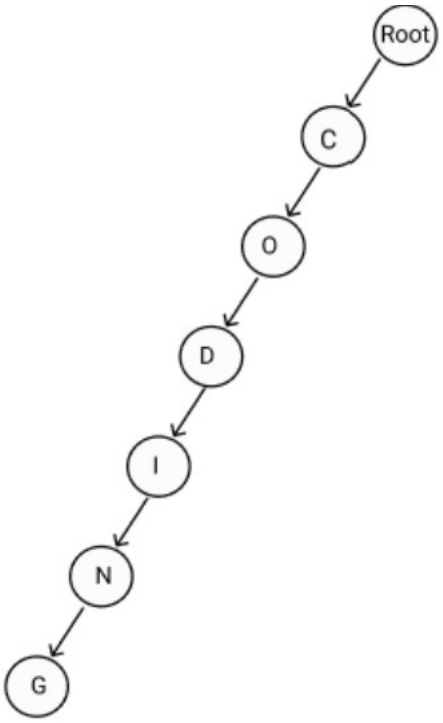
1
5
insert coding
insert ninja
countWordsEqualTo coding
countWordsStartingWith nin
erase coding

**Sample Output 1:**

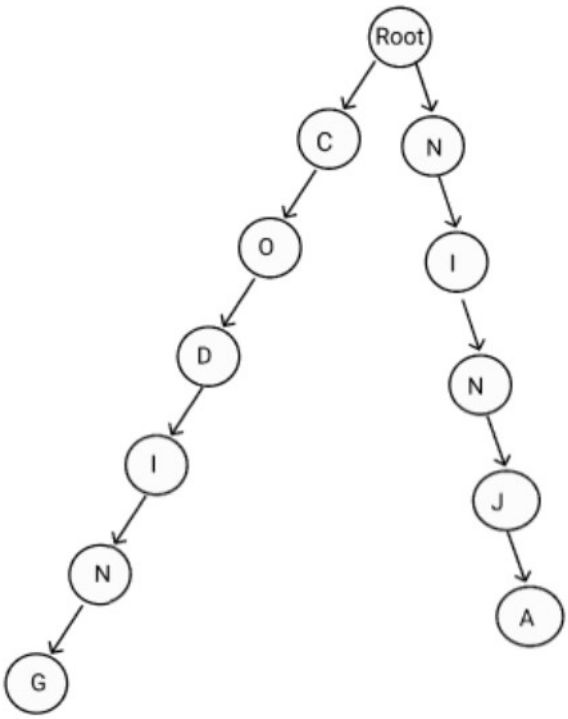
1
1

### Explanation Of Sample Input 1:

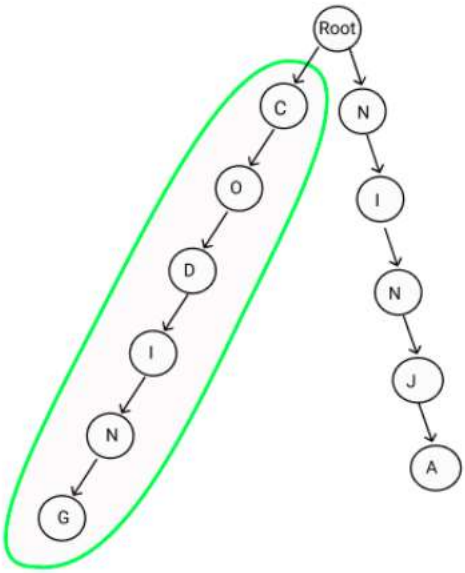
```
After insertion of “coding” in
“TRIE”:
```



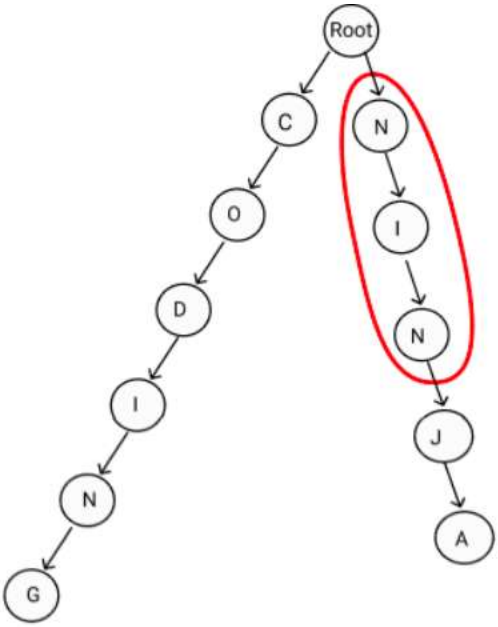

```
After insertion of “ninja” in
“TRIE”:
```



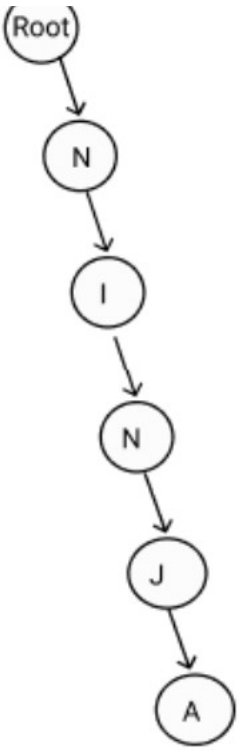

```
Count words equal to “coding” :
```




```
Count words those prefix is “nin”:
```




After deletion of the word “coding”, “TRIE” is:




**Sample Input 2:**

1
6
insert samsung
insert samsung
insert vivo
erase vivo
countWordsEqualTo samsung

countWordsStartingWith vi

**Sample Output 2:**

2

0

**Explanation for sample input 2:**

insert “samsung”: we are going to insert the word “samsung” into the “TRIE”.

insert “samsung”: we are going to insert another “samsung” word into the “TRIE”.

insert “vivo”: we are going to insert the word “vivo” into the “TRIE”.

erase “vivo”: we are going to delete the word “vivo” from the “TRIE”.

countWordsEqualTo “samsung”: There are two instances of “sumsung” is present in “TRIE”.

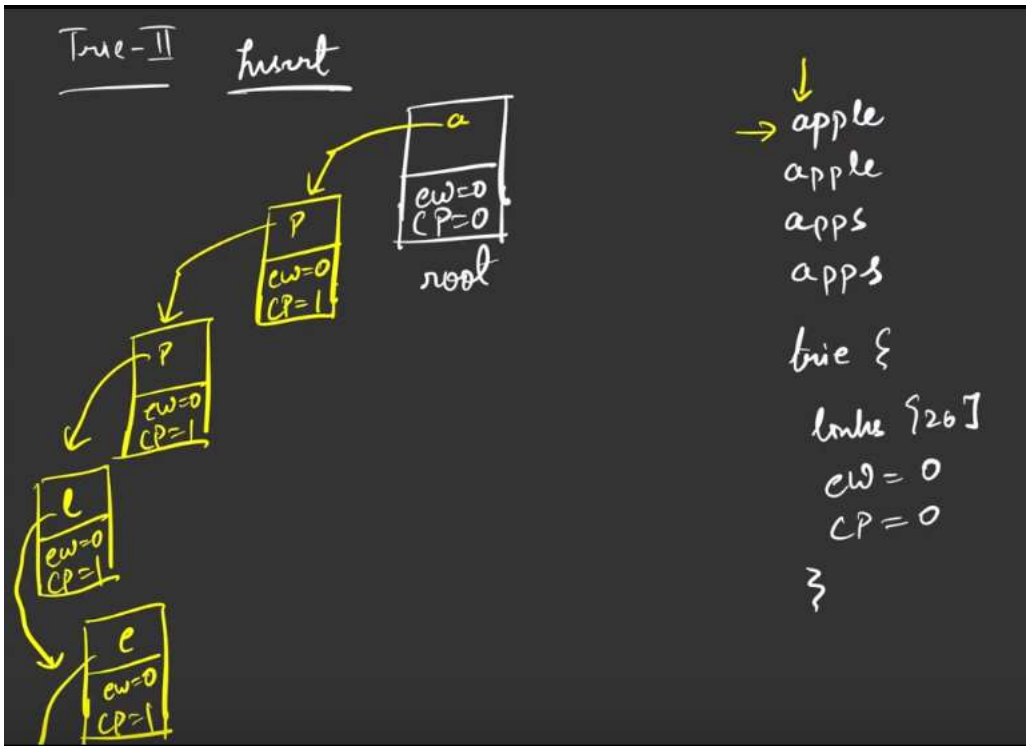
countWordsStartingWith “vi”: There is not a single word in the “TRIE” that starts from the prefix “vi”.

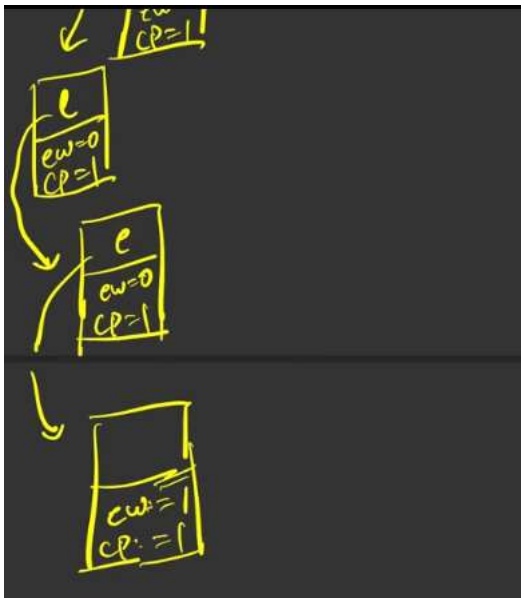
## Approach

This time our requirement is different so we need to modify our trie accordingly.
We will again have array of 26 but instead of bool flag, we will bave a variable endWith = 0, Variable countPrefix = 0.
We will initially declare a root with countPrefix (CP) = 0, endWith (EW) = 0

### Insert

We will insert "apple"
At every node, ew = 0, cp = +1

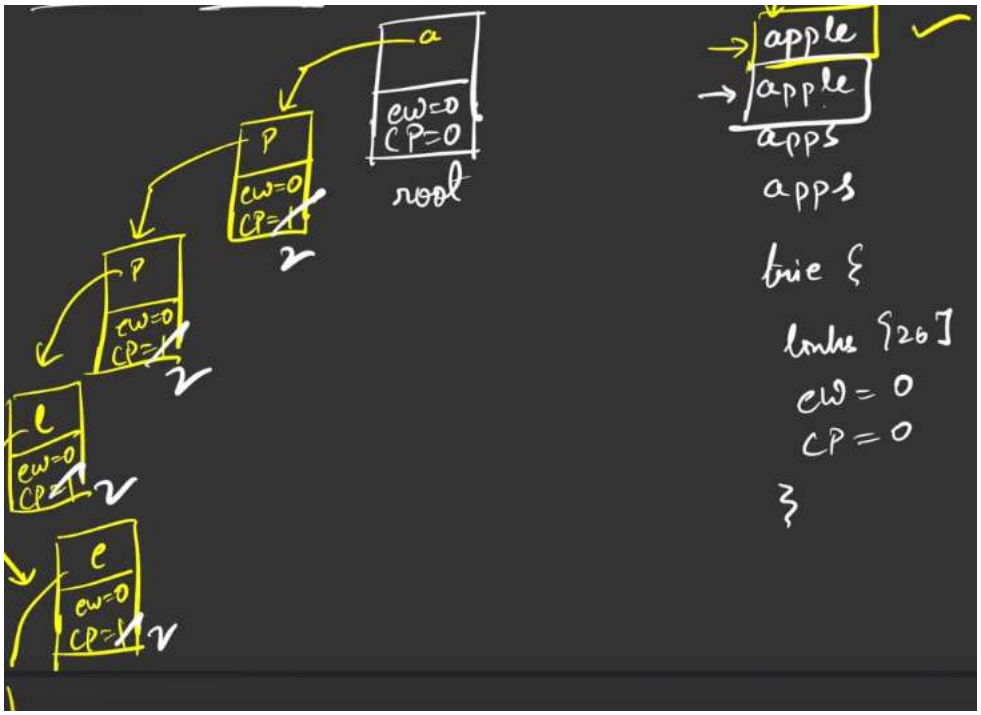




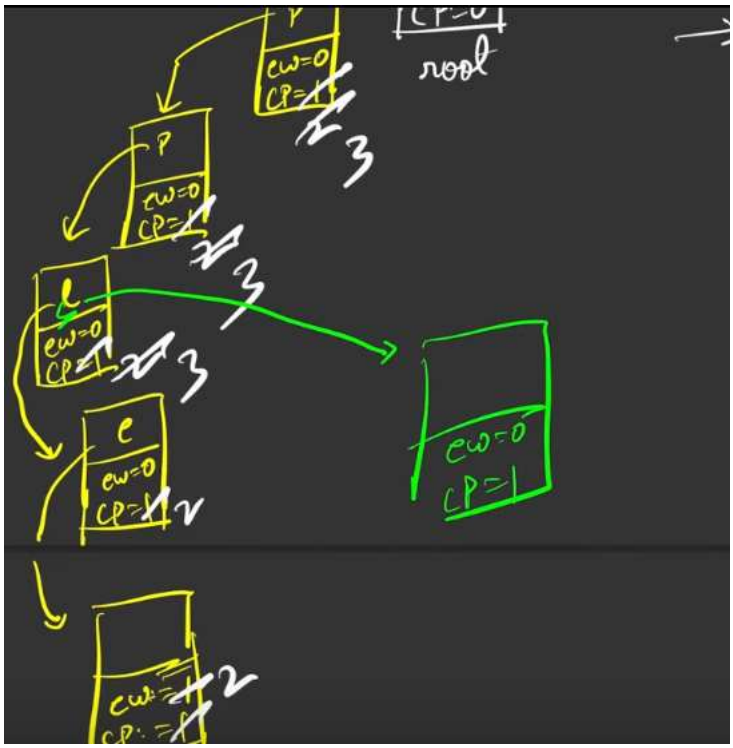

At the end node, ew = 1,

Now go for next test case "apple" in apple

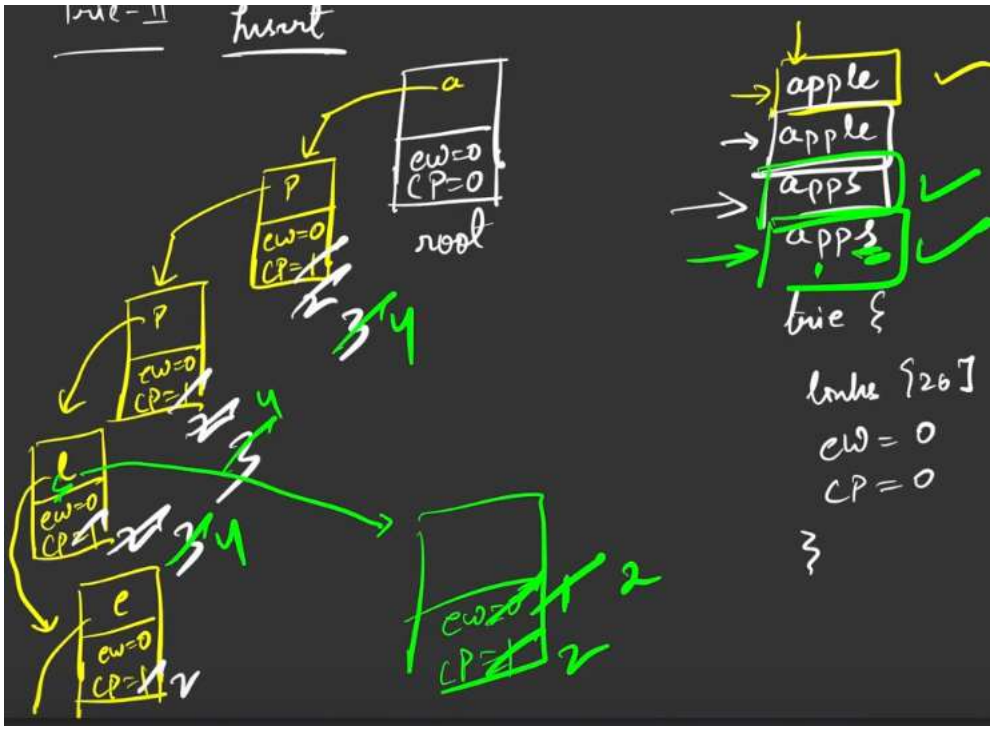
We see "a" yes it is there, so go to ref node of "a" i.e "p" and mark it CP = 1 se 2
Now we see "p" yes we found it, go to its ref node and make CP = 1->2
Now again we see "p" yes we found it, go to its ref node and make CP = 1->2
Now we see "l", we found do the same
Now we see "e", we found do the same with last node, ew = 1, cp = 2
Now we are at last node




We check for "apps", we increse CP = 3 for "a", "p", "p" but we did not found "s" so insert "s" in ref node of "p" with CP = 1, EW = 0
Now "apps" ends so make sure to make EW = 1 for "apps" last node




Now for "apps"




So for insert, we make a ref node and EW = 0, CP = 1
When we reach end we make EW to 1, CP is already 1

## Count Words End( )

Check character by character when you reach end of word, return EW of that last node
If a character does not exists then return false

## Count Words Start with( )

Check character by character and last character ka ref node ka EW will tell, number of word starts with( )

## Erase( )

Start from root, and as you found character by character, reduce the CP of ref node of each character.
When you reach end of word, reduce EW of last node also

## Code

```
struct Node{
    Node* arr[26];  // 0-based indexing
    int endWith = 0;
    int countPrefix = 0;

    bool containsLetter(char ch)
    {
        return (arr[ch - 'a']);
    }

    void put(char ch, Node* node)
    {
        arr[ch-'a'] = node;
    }

    Node* get(char ch)
    {
        return arr[ch - 'a'];
    }

    void incrementEndWith()
    {
        endWith++;
    }

    void decrementEndWith()
    {
        endWith--;
    }

    void incrementCountPrefix()
    {
        countPrefix++;
    }

    void decrementCountPrefix()
    {
        countPrefix--;
    }

    int getEndWithofLastNode()
```

```
    {
        return endWith;
    }

    int getCountPrefix()
    {
        return countPrefix;
    }

};

Class Trie{

    private:

    Node* root;

    public:

    Trie(){
        root = new Node;
    }

    void insert(string &word){
        Node* node = root;
        for(int i = 0;i<word.size(); i++)
        {
            if(!node->containsLetter(word[i]))
            {
                node->put(word[i],new Node());
            }
            node = node->get(word[i]);
            node->incrementCountPrefix();
        }
        // last node
        node->incrementEndWith();
    }

    int countWordsEqualTo(string &word){
        // start from root
        Node* node = root;
        for(int i = 0;i<word.size(); i++)
        {
            if(node->containsLetter(word[i]))
            {
                node = node->get(word[i]);
            }
            else{
                return 0;
            }
        }
        // End with has info of number of words ending with this word
        return node->getEndWithofLastNode();
    }

    int countWordsStartingWith(string &word){
        // start from root
        Node* node = root;
```

```
        for(int i = 0;i<word.size(); i++)
        {
            if(node->containsLetter(word[i]))
            {
                node = node->get(word[i]);
            }
            else{
                return 0;
            }
        }
        // Count Prefix has information of number of word whose prefix
 is this word
        return node->getCountPrefix();
    }

    void erase(string &word){
        // start from root
        Node* node = root;
        for(int i = 0;i<word.size(); i++)
        {
            // Assume we found it, so decrese CP for everyone
            if(node->containsLetter(word[i]))
            {
                node = node->get(word[i]);
                node->decrementCountPrefix();
            }
            else{
                return;
            }
        }
        // we are at last node
        // so decrease EW also
        node->decrementEndWith();
    }
};
```