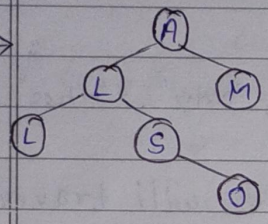
Enroll
Page No: 474
Date: 01/10/24

# TRIES

Retrieval Tree
or

* Trie is also known as a Prefix Tree because it is commonly used to store prefixes of strings or characters.
* A Trie (pronounced by "try") is a tree-like data structure.
* It is used for efficiently storing & searching a dynamic set of strings, where keys are usually words or sequences of charactors.

Ex →



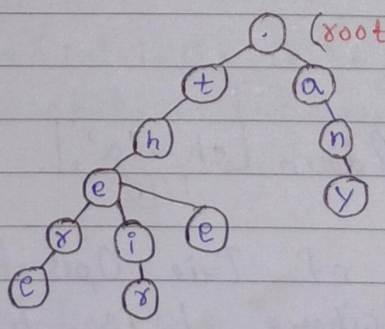

Prefix Tree

There — apple
The — app
Their — applet

→ prefix ←

* Tries can have more than two nodes.
* In Most of the Case, Tries height are smaller than Binary tree so Tries time complexity is efficient sometimes.
* Tries are mostly efficient for Strings.

Ex → words [ ] = "the", "a", "there", "their", "any", "thee"




# prefix बस एक बार store होता है।

## ★ CREATING A TRIE

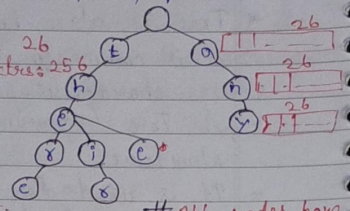
Words[] = "the", "a", "there", "their", "any", "thee".

Creation ⇒

```
class Node {
    Node[] children;
    boolean endOfWord;
}
```

In general use: 26 for all characters: 256

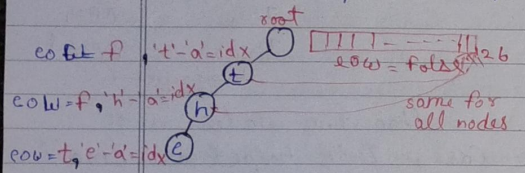
endOfWord: tells us that the word ends or not. → true / false



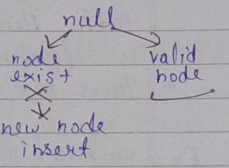

# All nodes have there fixed size

## ★ INSERT IN TRIE O(L)

Words[] = "the", "a", "there", "their", "any", "thee"




# We will traverse one by one on characters & check whether it exists or not.




Pseudo Code:

```
curr = root
for (level=0 to word.length)
    if (curr.children[ch-'a'] == null)        // idx
        → new node create
    else
        X
    curr = curr.children[ch-'a'];             // idx
```

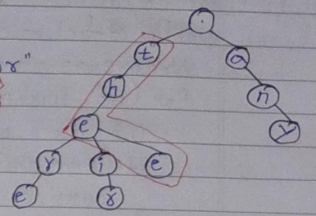
# Time Complexity of Trie Operations → O(L)

where, L is ~~level of Trie~~ Length of largest word.



## ★ SEARCH IN TRIE - O(L)

* Key = "thee"
  - all desirable char true.
  - eow = true.
* Key = "thor"



# Search will be complete when, if all letters exist & endOfWord = 'true'

Pseudo Code:

```
curr = root
for (level = 0 to key.length())
    idx = key.charAt(level) - 'a'
    if (curr.child[idx] == null)
        → false
    curr = curr.child[idx]
eow ⇒ curr
```

⇒

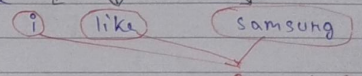
```
// Creating a Trie
static class Node {
    Node children[] = new Node[26];
    boolean eow = false;
    Node() {
        for (int i=0; i<26; i++) {
            children[i] = null;
        }
    }
}
public static Node root = new Node();
                          //root is Empty.
```

```
//Insert in Trie - O(L)
public static void insert (String word) {
  Node curr = root;
  for (int level=0; level < word.length(); level++) {
     int idx = word.charAt (level) - 'a';
     if (curr.children [idx] == null) {
        curr.children [idx] == new Node();
     }
     curr = curr.children [idx];
  }
  curr.eow = true;
}
//Search in Trie - O(L)
public static boolean Search (String key) {
  Node curr = root;
  for (int level=0; level < key.length(); level++) {
     int idx = key.charAt (level) - 'a';
     if (curr.children [idx] == null) {
        return false;
     }
     curr = curr.children [idx];
  }
  return curr.eow == true;
}
public static void main (String args []) {
  String words [] = {"the", "a", "there", "their", "any", "thee"};
  for (int i=0; i< words.length; i++) {
     insert (words [i]);
  }
  System.out.println (Search ("thee") ); // true
  System.out.println (Search ("thor")); // false
}
```



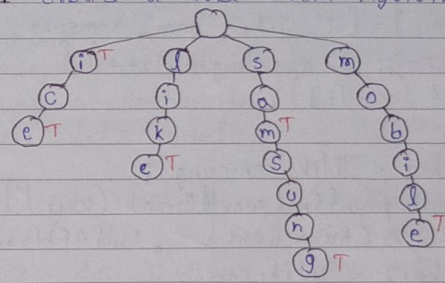
★ WORD BREAK PROBLEM - Given an input string & a dictionary of words, find out of the input string can be broken into a space-separated sequence of dictionary words.

Input: Words[] = {i, like, sam, samsung, mobile, ice}
Key = "ilikesamsung"
Output: true




Present in words[] → true

Approach: We will use Substrings Approach.
S1 - Create a trie from mystrings.




Key = "ilikesamsung" → for (i to key.length){

i=0

i | like samsung

EOW, SEARCH, TRUE

i | like samsung | i=0
il F | ikesamsung | i=1
lik F | esamsung | i=2
like | samsung | i=3
True

i=0

s | amsung
F

samsung → At last, EOW→T, SEARCH→T.

// Time Complexity - O(L) or O(n)

```
public static boolean wordBreak (String key) {
    if (key.length() == 0) { //Base Case
        return true;
    }
    for (int i=1; i<=key.length(); i++) {
        if (search(key.substring(0,i)) &&
            wordBreak(key.substring(i))) {
            return true;
        }
    }
    return false;
}
public static void main (String args[]) {
    String arr[] = {"i", "like", "sam", "samsung", "mobile", "ice"};
    for (int i=0; i<arr.length; i++) {
        insert (arr[i]);
    }
    String key = "ilikesamsung";
    System.out.println (wordBreak (key));
    //true -> Characters ✓, EndOfWord ✓
    String key2 = "ilikesam";
    System.out.println (wordBreak (key2));
    //true -> Characters ✓, EndOfWord ✓
    String key3 = "ilikesung";
    System.out.println (wordBreak (key3));
    //false -> Characters ✗, EndOfWord ✗
    String key4 = "ilik";
    System.out.println (wordBreak (key4));
}
    //false -> Characters ✓,
               EndOfWord ✗
```



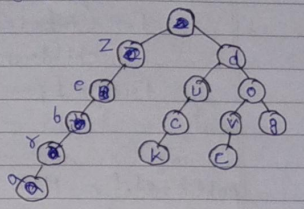
★ PREFIX PROBLEM – Find the shortest unique prefix for every word in a given list. Also, Assume no word is prefix of another.

Input: arr[] = {"zebra", "dog", "duck", "dove"}
Output: ans = {"z", "dog", "du", "dov"}

shortest prefix ⇓ unique

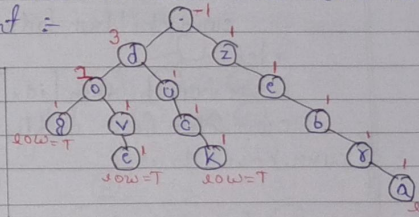
| dove | duck |
|---|---|
| d ✗ | d ✗ |
| do ✓ | du ✓ |

d is in both so it isn't unique.



S1: Create a Trie

S2: We will create a variable in class Node which will count a letter that in how much word it come. Let's count :-



# जहाँ frequency = 1 हो जाती है वहाँ तक Print करना होता है।

Ex → zebra, dog, dove
1 | 3 2 1 | 3 2 1
'z', "dog", "dov"

S3: insert() → if curr.child[idx] == null
new node;
else
frequency ++;

S4: root.frequency == 1 ⟶ print answer.

//O(L) = Level of Trie = Longest word of Trie

⇒
```
static class Node {
    Node [] children = new Node [26];
    boolean eow = false;
    int freq;
    public Node() {
        for (int i=0; i< children.length ; i++)
            children [i] = null;
        freq = 1;
    }
}
public static Node root = new Node();
public static void insert (String word) {
    Node curr = root;
    for (int i=0; i< word.length(); i++)
        int idx = word.charAt(i) -'a';
        if (curr.children [idx] == null)
            curr.children [idx] = new Node();
        else
            curr.children [idx].freq + +;
        curr = curr.children [idx];
    curr.eow = true;
}
public static void findPrefix (Node root, String ans) {
    if (root == null)
        return;
    if (root.freq == 1)
        System.out.println (ans);
        return;
    for (int i=0; i< root.children.length; i++)
        if (root.children [i] != null)
            findPrefix (root.children [i], ans+(char)
                                        (i+'a'));
}
```

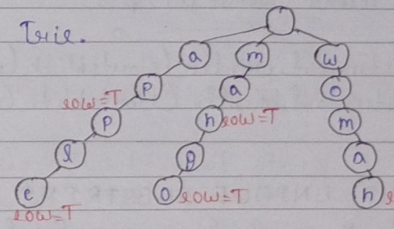


```
public static void main (String args[]) {
    String arr[] = {"zebra", "dog", "duck", "dove"};
    for (int i=0; i< arr.length; i++)
        insert (arr [i]);
    root.freq = -1;
    findPrefix (root, " ");  // Output will be in
}                               alphabetical order.
```

★ STARTSWITH PROBLEM — Create a function boolean startsWith (String Prefix) for a trie. Returns true if there is a previously inserted string word that has the prefix, & false otherwise.

Input: words[] = {"apple", "app", "mango", "man", "woman"}
prefix = "app" Output: true
prefix = "moon Output: false

SL: Create a Trie.




* In this, we don't have to check the condition of End-of-Word because we have to search only prefix of word.

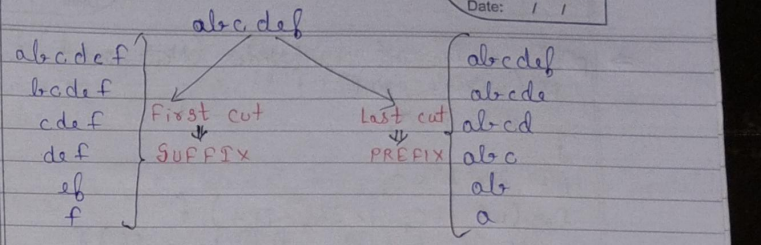
* We will check one-by-one character in our levels of Trie. — O (L)

⇒
```
public static boolean startsWith (String prefix) {
    Node curr = root;
    for (int i=0; i<prefix.length(); i++) {
        int idx = prefix.charAt(i) - 'a';
        if (curr.children[idx] == null) {
            return false;
        }
        curr = curr.children[idx];
    }
    return true;
}
public static void main (String args[]) {
    String words[] = {"apple", "app", "mango", "man", "woman"};
    String prefix1 = "app";
    String prefix2 = "moon";
    for (int i=0; i<words.length; i++) {
        insert (words[i]);
    }
    System.out.println (startsWith (prefix1));
    System.out.println (startsWith (prefix2));
}
```

★ COUNT UNIQUE SUBSTRINS – Given a string of length n of lowercase alphabets, we need to count total number of distinct substrings of this string. → str = "ababa"
→ ans = (10).

{"a", "b", "ab", "ba", "aba", "bab", "abab", "baba", "ababa", "" }
null string






* For Unique Substring = ① all prefix of all suffix =
② all suffix of all prefix.

* Imagine first case, so firstly write down all suffix of "ababa" :-

| SUFFIX | UNIQUE PREFIX |
|---|---|
| ababa → | a ab aba abab ababa |
| baba → | b ba bab baba " " |
| aba → | ~~a ab aba~~ |
| ba → | ~~b ba~~ |
| a → | ~~a~~ |

↓ 10.

# Trie के अंदर Unique Prefixes होते हैं।

Approaches S1: Find all suffix of string
S2: Create TRIE + insert
S3: count nodes of Trie

* For suffix, & insert Trie → for (int i=0 to n) { str.substring(i) insert Trie ( ) }

```
⇒ public static int countNodes (Node root) {
    if (root == null) {
        return 0;
    }
    int count = 0;
    for (int i=0; i<26; i++) {
        if (root.children [i] != null) {
            count += countNodes (root.children [i]);
        }
    }
    return count + 1;
}
public static void main (String args []) {
    String str = "ababa";
    //suffix -> insert in trie
    for (int i=0; i<str.length(); i++) {
        String suffix = str.substring (i);
        insert (suffix);
    }
    System.out.println (countNodes (root));
}
```

★ LONGEST WORD WITH ALL PREFIXES

– Find the longest string in words such that every prefix of it is also in words.

Input: words = ["a", "banana", "app", "appl", "ap", "apply", "apple"]

Output: ans = "apple". # also ans can be "apply" but we want smaller Lexicographic.

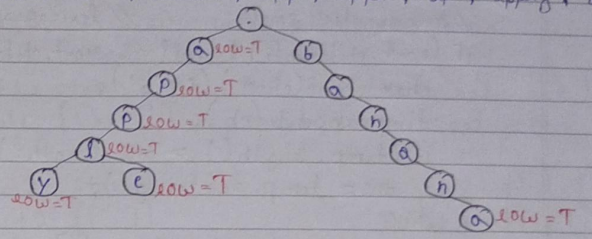


Case 1: String → single answer

Case 2: strings → small lexicographic

Case 3: doesn't exist → " "

Approach: words = ["a", "banana", "app", "appl", "ap", "apply", "apple"]




★ We have to find that Longest word in TRIE ✓whose all characters eow = true.

```
(temp len > ans len){
    update;
}
→ String final ans = "apple"   { String temp में जो word आयेगी वो Update हो जायेगी final ans में.
→ String temp = a+p+p+l+e
Backtracking → for (root → children) {
(at last temp will & be empty.)   backtrack   root != null
                                              eow = T
→ print String final ans.
```

# By Lexicographic, "e" will come first in comparision with "y".

Pseudo Code →

```
void longestWord (root, temp)
    if (root == null) { return; }
    for (int i=0 to 26)
        if (root.child [idx] != null && child [idx].eow == true)
            temp + add char (root)
            if (temp len > ans len)
                ans = temp
            longestWord (root.child [i], temp)
```

temp → delete for last character. → backtrack

⇒
```
public static String ans = "";
public static void longestWord(Node root, StringBuilder temp){
    if(root == null)
        return;
    for(int i=0; i<26; i++) or for(int i=25; i>=0; i--)
        //lexicographic small          // lexicographic large
        if(root.children[i] != null && root.children[i].eow == true){
            char ch = (char)(i+'a');
            temp.append(ch);
            if(temp.length() > ans.length()){
                ans = temp.toString();
            }
            longestWord(root.children[i], temp);
            temp.deleteCharAt(temp.length()-1); //backtrack
        }
    }
}

public static void main(String args[]){
    String words[] = {"a", "banana", "app", "appl",
                      "ap", "apply", "apple"};

    for(int i=0; i<words.length; i++){
        insert(words[i]);
    }
    longestWord(root, new StringBuilder(""));
    Syso(ans);
}
```

Output : apple.



Ques 1: Group Anagrams Together — Given an array of strings strs, group the anagrams together. We can return the answer in any order.

→ An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Input 1: strs = ["eat", "tea", "tan", "ate", "nat", "bat"]
Output 1: [["bat"], ["nat", "tan"], ["ate", "eat", "tea"]]

Input 2: strs = [""]
Output 2: [[""]]

Input 3: strs = ["a"]
Output 3: [["a"]]

⇒
```
class TrieNode{
    List<String> data;
    TrieNode children[];
    boolean isEnd;
    TrieNode(){
        data = new ArrayList<>();
        children = new TrieNode[26];
        isEnd = false;
    }
}
class Solution{
    static TrieNode root;
    List<List<String>> ans;
    public List<List<String>> groupAnagrams(String args[])
        ans = new ArrayList<>();
        root = new TrieNode();
        for(String word : args){
            build(word);
        dfs(root);
        return ans;
}
```

```
public void build (String s) {
  TrieNode temp = root;
  char[] word = s.toCharArray();
  Arrays.sort (word);
  for (char c: word) {
    TrieNode child = temp.children (c-'a');
    if (child == null) {
        temp.children[c-'a'] = new TrieNode();
        temp = temp.children[c-'a'];
    }
  temp.isEnd = true;
  temp.data.add(s);
}
public void dfs (TrieNode rt) {
  if (rt.isEnd)
    ans.add (rt.data);
  for (int i=0; i<26; i++)
    if (rt.children[i] != null)
        dfs (rt.children[i]);
}
```

Ques 2. Longest word in Dictionary - Given an array of strings words representing an English Dictionary, return the longest word in words that can built one character at a time by other words in words.

→ If there is more than one possible answer, return the longest word with the smallest lexicographical order.

→ If there is no answer, return the empty string.

→ Note that the word should be built from left to right with each additional character being added to the end of a previous word.



⇒

```
private static class Node {
  private char data;
  private String word;
  private boolean isEnd;
  private Node[] children;
  public Node (char data) {
      this.data = data;
      this.word = null;
      this.isEnd = false;
      this.children = new Node [26];
  }
}
private Node root = new Node ('/');
private String ans = "";
private void insert (String word) {
  Node curr = this.root;
  for (int i=0; i<word.length(); i++)
    int childIdx = word.charAt (i) - 'a';
    if (curr.children [childIdx] == null)
        curr.children [childIdx] = new Node (word.charAt(i)
    curr = curr.children [childIdx];
  curr.isEnd = true;
  curr.word = word;
}
public String longestWord (String[] words) {
  for (String word : words)
      insert (word);
  Node curr = this.root;
  dfs (curr);
  return ans;
}
```

```
private void dfs (Node node) {
  if (node == null) {
    return;
  }
  if (node.word != null) {
    if(node.word.length() > ans.length()) {
      ans = node.word;
    }
    else if (node.word.length() == ans.length && node.word.compareTo(ans)<0)
      ans = node.word;
    }
  }
  for (Node child : node.children) {
    if (child != null && child.word != null) {
      dfs (child);
    }
  }
}
```

Input: words = ["a", "banana", "app", "appl", "ap", "apply", "apple"]
Output: "apple"

# Both "apply" & "apple" can be built from other words in the dictionary. However, "apple" is lexicographically smaller than "apply".