21 Wednesday September
39th Wk • 264-101

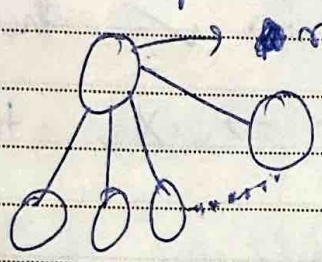
2022 SEPTEMBER

M	T	W	T	F	S	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		

Trie

There are several other ~~[illegible]~~ names to Trie data structure - prefix, digital search, Retrieval tree, etc.

Trie is a K-ary tree. A K-ary tree is a tree which can have multiple nodes or K nodes.



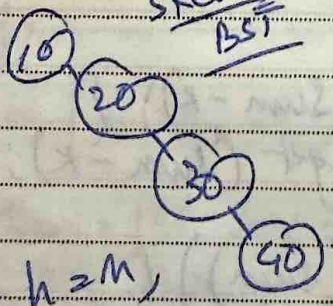
whereas, ~~[illegible]~~ in a binary tree we have maximum no. of 2 nodes for a root.

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

If there is a String array given, then we prefer to use Trie data structure, because the search optimization is better than Binary Search Tree.

10, 20, 30, 40, ...

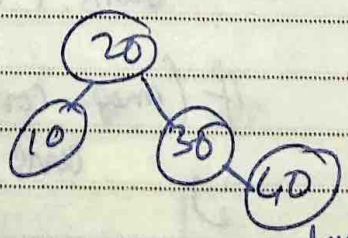
BST



skewed BST

Search
= O(h), where h = n,
In worst case scenario, skewed BST, h = n and time complexity goes to O(n).

AVL



In this type of tree, time complexity = O(h), h = log n.

Compared to these BST, Trie offers a better time complexity = $O(L)$, where L = word length.

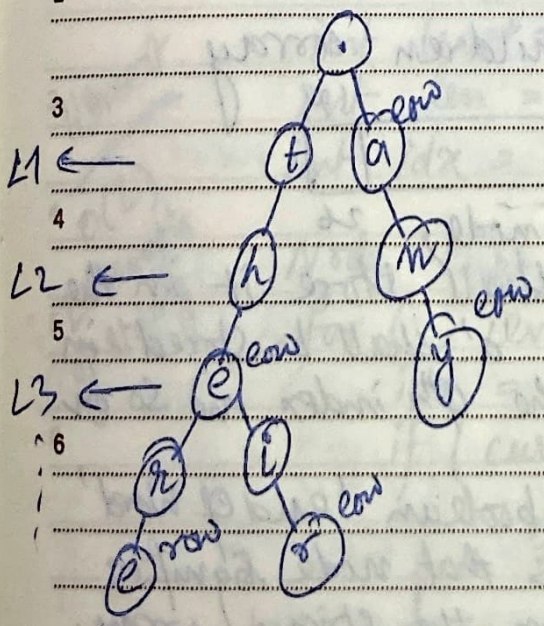
characteristics of Trie :-

(i) Search complexity = $O(\log n)$, so it offers a fast time complexity.

(ii) root is an empty node. It doesn't contain any data but it just directs to the children.

(iii) Prefix is not repeated, meaning suppose if I have "apple", "app" then we shall store "ap" only once.

E.g, words[] = "the", "a", "there", "their", "any"



✓ words are stored letter by letter in Trie.

✓ once a words is finished, we shall mark the node by 'eow', meaning 'end of word'.

✓ once, we start to store a letter, we shall check whether that letter is present in a level. If yes, then we shall check the child node for the next letter.

For. e.g, for "there", we check (t)-(h)-(e), then for 'r', we create a new child node and store it.

23 September
39th Wk • 266-099
Friday

2022 SEPTEMBER

M	T	W	T	F	S	S	M	T	W	T	F	S	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									

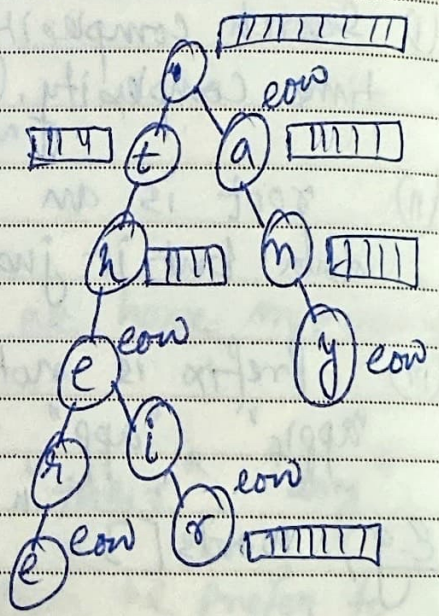
```
class Node {
    Node[] children;   // 26
    boolean endOfWord;
}
```

children → variable

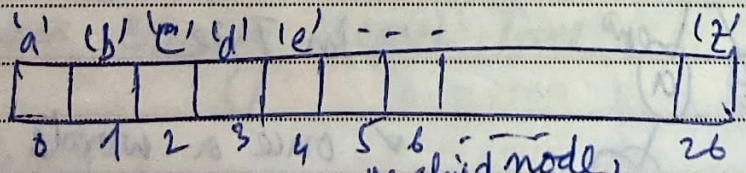
```
static class Node {
    int data;
    Node left;
    Node right;
}
```

→ If 'a' to 'z' is required to be stored, then 26 shall be the size.

→ If 'a' to 'z', then 'A' to 'Z', and then '@', '!', '$', such all characters need to be stored, then 256 shall be the size.



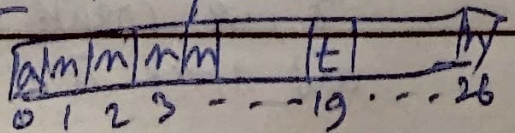
→ Now, every node shall have a children array.



So, whatever the letter is in the child node, we shall store it in the corresponding index. So, 'a' shall be stored in the 0th index, 'c' shall be in the 2nd index, 'e' shall be in the 4th index and so on.

→ For every node, we check the boolean 'endOfWord'. If it's true, then that means that node signifies the end of a complete word from the string array. If false, that means no word ends there.

→ Note: In this example, the children array of the root node is —



n → null.

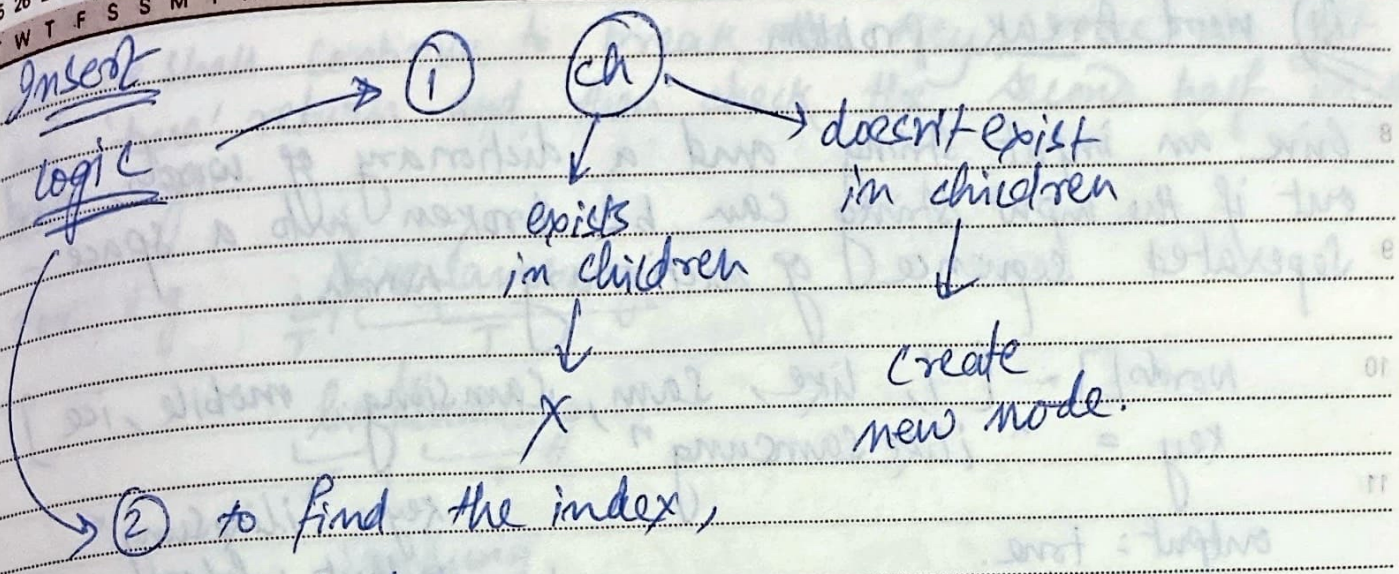
code :- Implementation, Insert & search.

```
8 → Public class Tries {
      static class Node {
9       Node [] children = new Node[26];
        boolean eow;
10
        Public Node() {
11        for (int i=0; i<26; i++) {
              children[i] = null;
12      }
      }
    }
1   Public static Node root = new Node();

2   Public static void insert (String word) { //o(n)
      int level = 0;
3   //insert  int len = word.length();
      int idx = 0;
4   o(L) L=length of string.  Node curr = root;
5     for (; level < len; level++) {
        idx = word.charAt(level) - 'a';
6       if (curr.children[idx] == null) {
          curr.children[idx] = new Node();
        }
        curr = curr.children[idx]; //update root
      }
      curr.eow = true; // if (level == word.length()-1)
    }
    Public static boolean search (String key) { //o(n)
//search  int level = 0;                    int idx = 0;
      int len = key.length();
```

```
Node curr = root;

for(; level < len; level++) {
    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(String word : words) {
    insert(word);
    System.out.println("inserted" + word);
}

System.out.println("thee →" + search("thee"));
System.out.println("thor →" + search("thor"));

System.out.println(startsWith("the"));
System.out.println(startsWith("thi"));
}
}
```

if(level == key.length() - 1
&& curr.eow == false)
return false;

Insert logic



② to find the index,

'a' - 'a' = 0

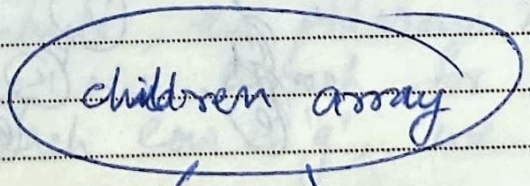
'b' - 'a' = 1

'c' - 'a' = 2 --- and so on.

idx = word.charAt(level) - 'a';

③ if(i = word.length() - 1)
root.eow = true;

Search logic




```
for(int i=0 to Key.length()) {
    idx = Key.charAt(i) - 'a'
    if (root.children[idx] == null)
        return false;
    if (i == Key.length() - 1
        && root.children[idx].eow == false)
        return false;
}
return true;
}
```

27 September
40th Wk • 270-095
Tuesday

2022 SEPTEMBER

M	T	W	T	F	S	S	M	T	W	T	F	S	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									

Q1) Word Break problem

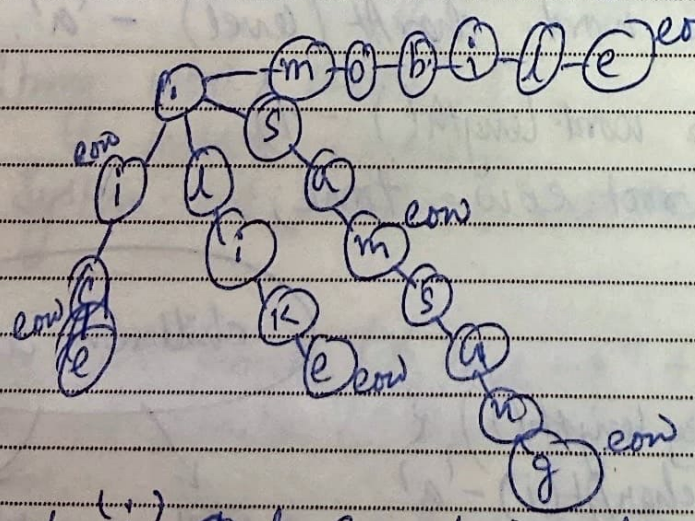
Give an input string and a dictionary of words, find out if the input string can be broken into a space-seperated sequence of dictionary words.

words[] = { i, like, sam, samsung, mobile, ice }
key = "ilikesamsung"
output: true.

key = "ilikesung"
output: false

Logic
(1) Prepare a trie data structure with the words array.
(2) cut in every possible of the key and find it in the trie.

For eg,



now, cut at 'i' & search in trie. yes, 'i' exists and it's also 'eow'.

Now, cut at 'l' & search in trie. yes 'l' exists but not eow. So, cut at 'i' & search -- and so on. ultimately, cut at 'e' and search - yes 'e' exists and eow. So, 'i' and 'like' both cases are true.

Now, repeat same thing for all other characters.

Also, we shall continue to break the key, once we get the 'true' return and then check the second half in the same logic.

For e.g i|likeSamsung
T T

like|SamSung
T T

SamSung
T

→ when we get 'i' as eow, we break the key into 2 parts - 'i' & 'likeSamSung'.

→ when we get 'like' as eow, we break the rest of the key as another half.

→ Now, 'SamSung' is the part remaining. we get 'Sam' and eow. But if we break, we don't get 'Sung' in any search next. So, we conclude, the previous break was incorrect. And we continue our search for the whole part of 'SamSung'. once we reach eow in 'g', we return true for 'SamSung'.

So, in the above step 'like' is True and 'SamSung' is also true. Then, we send the 2 parts in the above step as 'true'. Then, we see both the parts in the 1st breakage are 'true'.

'i' → true, 'likeSamSung' → true.

Then, we can conclude that a possible key exists in the trie data structure.

29 September
40th Wk • 272-093
Thursday

2022 SEPTEMBER

M	T	W	T	F	S	S	M	T	W	T	F	S	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									

Pseudo code :-

```
bool wordBreak (String key) {
    if (key.length == 0)   // if length of the second half becomes zero.
        return true;
    for (int i = 1 to n) {
        1st part key(0, i);   → (0 to i-1)
        2nd part key(i, n);   → (i to n-1)
        if ( search (1st part) && wordBreak (2nd part))   → search for the 1st part
            return true;                                  → recursive call for the 2nd part
    }
    return false;
}
```

Java code :-

```
Public static boolean wordBreak (String key) {
    if (key.length() == 0) {
        return true;
    }
    for (int i = 1; i <= key.length(); i++) {
        String firstPart = key.substring (0, i);
        String secPart = key.substring (i);
        if (search (firstPart) && wordBreak (secPart)) {
            return true;
        }
    }
    return false;
}
```

```
Public static void main (String args[]) {
String words[] = {"i", "like", "sam", "samsung", "mobile"};
String key = "ilikeSamsung";
for (int i=0; i< words.length; i++) {
    insert (words[i]);   // insert function already written.
}
System.out.println ( wordBreak (key));
}
```

82) 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 prefix and false otherwise.

words[] = {"apple", "app", "mango", "man", "woman"}

prefix = "app", output: true
prefix = "moon", output: false.

→ This problem looks very similar to our search functionality, however we shall not try to impose the 'eow' part.

pseudo code

```
Node curr = root,
for (int i=0 to prefix.length())
    idx = prefix.charAt(i) - 'a';
if ( curr.children[idx] == null)
    return false;
curr = curr.children[idx];
}
return true;
```

NOTES

Java code :-

```
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 prefix = "app";
    for (int i=0; i< words.length ; i++) {
        insert (words[i]);
    }
    System.out.println (startsWith (prefix));
}
```

Note :- Insert search, node creation and implementing node functions are already written. (Pg - 278/279)

Action Plan

OCT '22

01 Sat

02 Sun

03 Mon

04 Tue

05 Wed

06 Thu

07 Fri

08 Sat

09 Sun

10 Mon

11 Tue

12 Wed

13 Thu

14 Fri

15 Sat

16 Sun

17 Mon

18 Tue

19 Wed

20 Thu

21 Fri

22 Sat

23 Sun

24 Mon

25 Tue

26 Wed

27 Thu

28 Fri

29 Sat

30 Sun

31 Mon

83) count unique Substrings

Given a string of length n of lowercase alphabet characters, we need to count total number of distinct substrings of this string.

str = "ababa", ans = 10.

Possible Sub-Strings = ~~a, b, ab, ba, a~~

a	ab	aba	abab	ababa
b	ba	bab	baba	
a	ab			
b	ba			
a				

But, we shall only chose unique substrings

→ a, b, ab, ba, aba, bab, abab, baba, ababa,

So, Count = 10.

→ To solve the problem, we shall use the concept of prefix - Suffix Concept.

→ Substring of a String

✓ All prefixes of all Suffix,

✓ All Suffixes of all prefix.

we shall use this method to solve the problem

OCT " "

01 October
40th Wk • 274-091
Saturday

2022 OCTOBER

M	T	W	T	F	S	S	M	T	W	T	F	S	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						

→ So, what we shall do is, firstly, we would find out all the suffixes and find prefixes of each suffix.

Str = "ababa"

Suffix

ababa → a, ab, aba, abab, ababa

baba → b, ba, bab, baba

aba → ~~a~~, ~~ab~~, ~~aba~~ [all are repeated]

ba → ~~b~~, ~~ba~~

a → ~~a~~

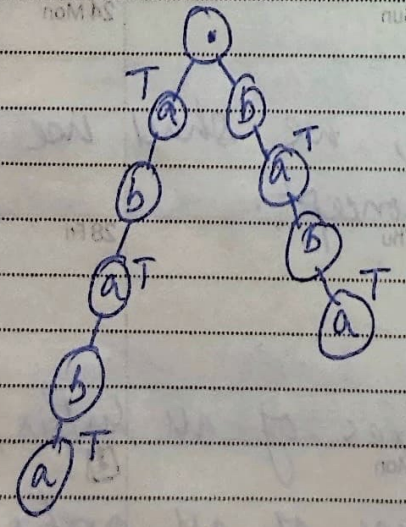
" " → " ".

So, count of all unique prefixes = 10.

steps → ① find all suffix of string
② create a trie from suffix
③ find out all unique prefix.

step ③ is simple - because,

total nodes of trie = count of unique prefix.



count of nodes = 10.

Java code :-

```
Public static int countNode (Node root)
{
    if (root == null) {    // base case
        return 0;
    }
    int count = 0;
    for (int i=0; i< 26; i++) {
        if (root.children[i] != null) {
            count += countNode (root.children[i]);
        }
    }
    return count + 1;    // (count+1) for my current node.
}

Public static void main (String args[]) {

String str = "ababa";

for(int i=0; i<str.length(); i++) {

    String suffix = str.subString (i); // all suffix
    insert (suffix); // insert into trie all suffix
}
    System.out.println( countNode (root));
}
```

(Q4) Longest Word with all Prefixes

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

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

ans = "apple".

→ In this example, both 'apple' and 'apply' have 5 characters and could have been the answer.
In these cases, where there are multiple answers, we shall choose the lexigraphically smaller answer.

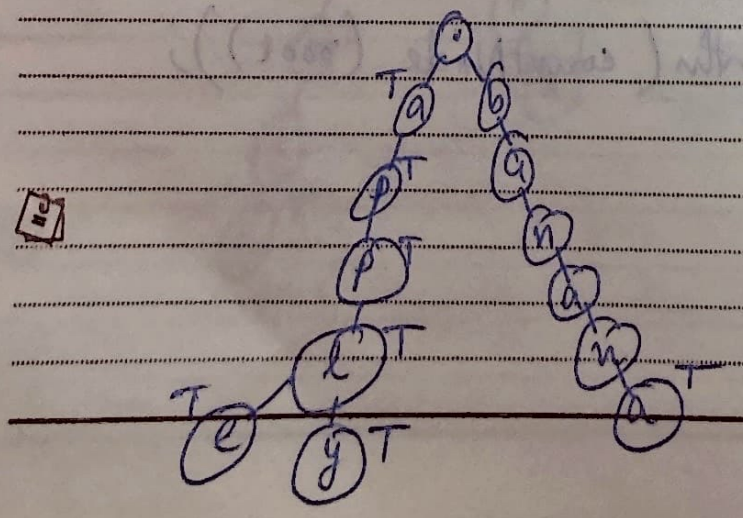
apple apply , e < y, so we choose apple.

So, there can be 3 cases to this problem.

Case 1 :- longest length word.

Case 2 :- equal length → choose lexicographically smaller.

Case 3 :- no match → empty string.



(T → endOfWord is true)

Java code :-

```
→ Public static String ans = "";

Public static void longestWord (Node root, StringBuilder temp)
{
    if (root == null) {
        return;
    }
    for (int i=0; i<26; i++) {
    if (root.children[i] != null && root.children[i].eow == true)
    {
        temp.append ((char) (i + 'a'));
        if (temp.length() > ans.length()) {
            ans = temp.toString();
        }
    longestWord (root.children[i], temp);
        temp.deleteCharAt (temp.length() - 1);
    }
}}
Public static void main (String args[]) {
String words[] = {"a", "banana", "app", "appl", "apply", "apple"};

for (int i=0; i< words.length; i++) {
    insert (words[i]); // 'insert' already scripted.
}
longestWord (root, new StringBuilder (Str: ""));
System.out.println (ans);
}
}
```