Trie Data Structure

(1)

» Efficient for the following operations on words in a dictionary.

→ Search
→ Insert
→ Delete
→ Prefix search
→ Lexicographical ordering of words.

(2)

I/p: insert("geek"), insert("geeks"), insert("bad"), insert("bat")
Search("bad"), search("eek"), print()

O/p
↳ Yes, No

bad, bat, geeks, geek

(3) Hashing vs Trie

» Prefix Search and lexicographic not supported in hashing, as hashing doesn't maintain keys in ordered form.

» Search, Insert and Delete

↙ Trie worst case is O(word_len).

↘ Hashing avg case is O(word_len).

(4) Java Implementation

» class TrieNode
{ TrieNode [] child = new TrieNode [26];
boolean isEnd;
}

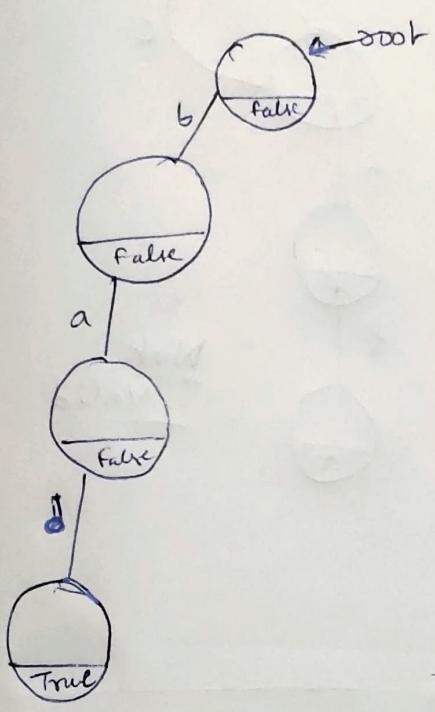
→ Every trie structure has 2 things :-

1) An array of children

2) A variable isEnd initialised to false.

→ Example representation.

dict = { bad, bat, geek, geeks }

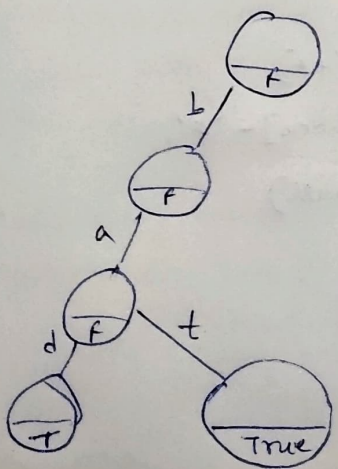


Inserting bad.

* isEnd might be true for non-leaf nodes also.

Eg:- Try inserting 'bads' in this tree.

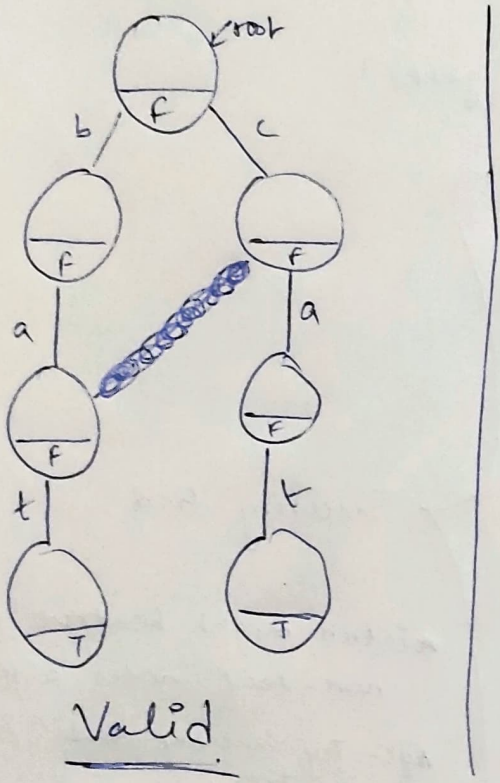
→ Inserting bat → we already have 'b' and 'a' so we directly go to these two child and add 't'.



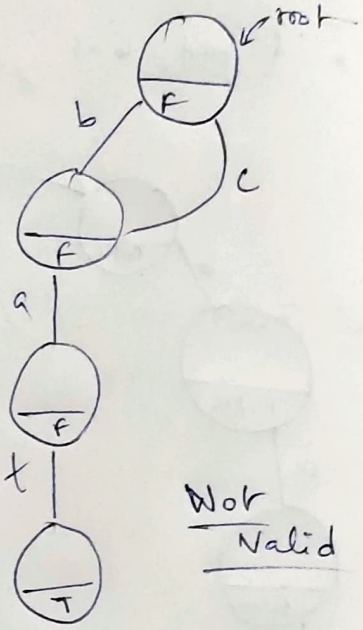
for bat addition.

Note :- If we have 'cat' as an input and 'bat' is already formed. We can't used 'at' of 'bat' for 'cat'.

Only children of C can be used as already existing values.



Valid.



Not Valid

Java implementation of Search.

Insert and Search is constant n, O(n).

n → key.length.

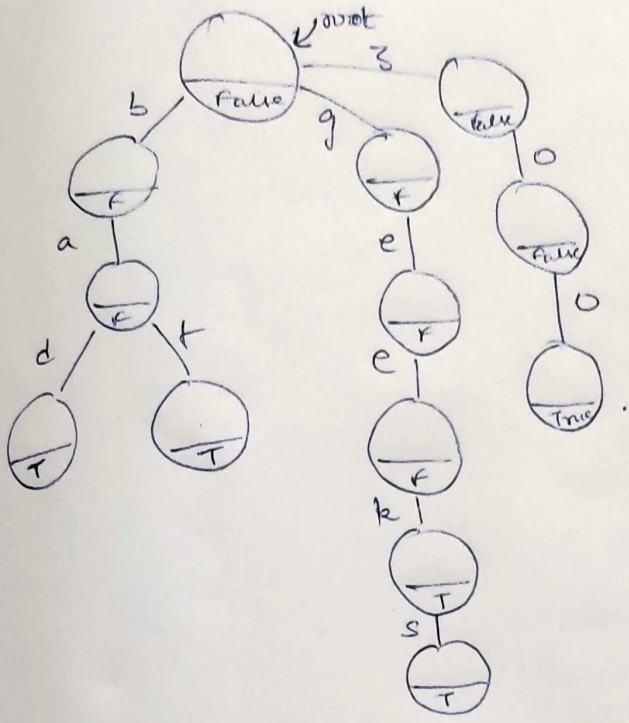
```
TrieNode root;

boolean Search(String key)
{ TrieNode curr = root;
  for(int i=0; i<key.length(); i++)
    { int index = key.charAt[index] - 'a';
      if(curr.child[index] == null)
        {return false;} *1
      curr = curr.child[index];
    }
    return curr.isEnd; *2
}
```

Java implementation of Insert

→ *1 → curr.child[index] = new TrieNode();

*2 → curr.isEnd = true;

Java implem of delete O(n) n → key length:



Case 1:
Deleting '300' will mean deleting whole branch.

Case 2:
Deleting 'geek' will mean just making k.isEnd = false.

Case 3:
Deleting 'bad' will mean deleting d. node/child.

```
TreeNode delNode (TrieNode root,
                  String key, int i)
{ if(root == null) return null;
  if(i == key.length)
  { root.isEnd == false;
    if(isEmpty(root) == true)
    { root = null; }
    return root;
  }
  int index = key.charAt(i)-'a';
  root.child[index] = delKey(root,
                  child[index]
                  , key, i+1);
  if (isEmpty(root) && root.isEnd == false)
  { root = null; }
  return root;
}

boolean isEmpty (TrieNode root)
{ for(int i=0; i<26; i++)
  { if(root.child[i] != null)
    { return false; }
  }
  return true;
}
```