Tries

- It is a tree based data structure, which is used for efficient retrieval of a key in a large data-set of strings.

- In trie, node's position in the tree defines the key with which it is associated and the key are only associated with the leaves.

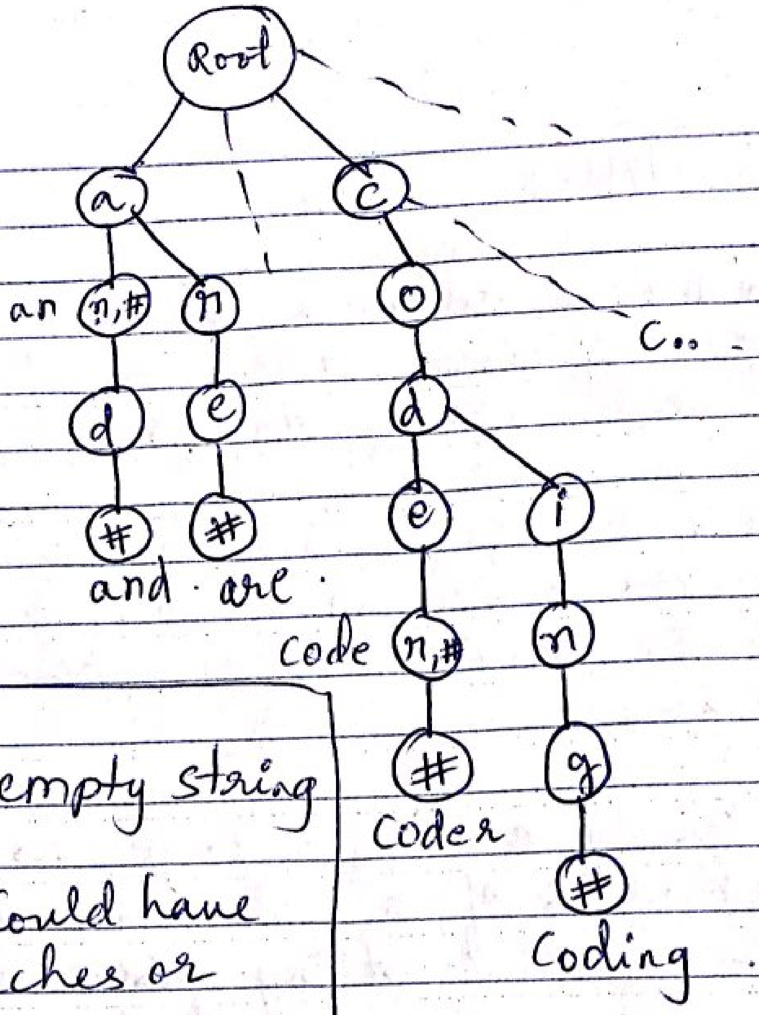
- Also known as Prefix tree as all descendants of a node have a common prefix of the string associated with that node, and root is associated with an empty string.

- Basic form of trie representation is that,

  → It is a linked set of nodes ~~where~~

  → where each node has an array of child pointers, one for each symbol in the alphabet (so for english, ⇒ 26 child pointers & for alphabet of bytes it is 256 pointers)

  → Trie node has flag which specifies whether it corresponds to the end of the key or not.



Root ⇒ empty string

~~*chars~~

- - - - ⇒ Could have more branches or pointers.

why Tries

* They can insert and find strings in $\boxed{O(L)}$ time (where L represents the length of a single word).

* This is much faster than hash table & ~~bin~~ BST.

Trie implementation :

~~Trie Node~~ 1) Trie Node

2) Search string

3) ~~S~~ Insert string

Insertion

- Every char of i/p key is inserted as an individual Trie Node.
- children is an array or list of pointers to next level trie nodes.
- key char acts as an index into the array children.
- If input key is
  - → new or an extension of existing key ⇒ construct ~~non ex~~ new nodes of the key & mark end of the word for last node.
  - → a prefix of the existing key in Trie, mark the last node of the key as the end of a word.
  - ~~→ Key leng~~
- Key length determines Trie depth.

Searching

- ull^{ler} to insert operation, ~~bu~~ however we only compare the characters and move down.
- We can terminate the search, due to the end of a string or lack of key in the trie.
- ~~'isEndOf~~

Issues with Tries representation

- They need a lot of memory for storing the strings
- For each node we have too many nodes
- They are faster but needs huge memory.

There are tries Compression techniques. ~~Bnd~~ to Improve tries representation, but these only reduce the memory at leaves but not at the internal nodes.

- ~~CTC~~

Implementation

```
class TrieNode {
    boolean isEndOfString;
    TriNode trieNodes ~~= new TrieNode[26];~~
    char data;
    public TrieNode(char c){
        this.data = c;
        thie.trieNodes = new TrieNode[26];
        isEndOfString = false;
    }

    public TrieNode subNode(char c) {
        if(trieNodes != null) {
            for(TrieNode trieNode: trieNodes){
                if(trieNode[c-'a']
                    != null) {
                    return trieNode[c-'a'];
                }
            }
        }
    }
}
class Trie {
```

CTCI Trie info .

- Trie is a variant of an n-ary tree. in which chars are stored at each node

Hash table vs trie .

- A hash will give O(1) for lookup & insertion, but ~~we~~ ~~hav~~ first we have to calculate the hash based on the i/p string (s) which is ~~ag~~ O(s).

In case of trie, insertion & lookup is linear with the length of i/p string s i.e., O(s) .

∴ In both cases asymptotic TC is linear .

we use hash table, when we need to lookup for full words .
This is easier to code, test & mainta

when we need to find prefixes or Suffixes use trie.