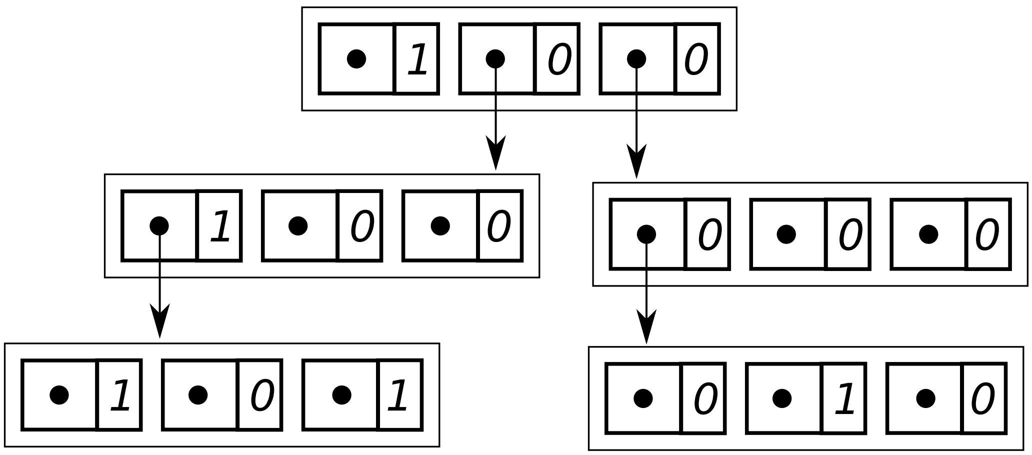
[python3] The *Trie* data structure is a kind of tree, and has similar intents as a binary search tree. It is used to store a set of keys, and then, one is able to efficiently query for whether a key is in the set or not.

A Trie works as follows. We assume that every key we want to store is a (non-empty) string in some alphabet, Σ. E.g., suppose $\Sigma = \{a, b, d\}$, and the set of strings we want to store in our data structure, i.e., Trie, is $\{a, ba, baa, bad, dab\}$. In the tree that is the Trie, each node corresponds to the alphabet, in this example, $\{a, b, d\}$. Each node can have as many children as the size of the alphabet; between 0 and 3 children, in our example. The path from the root to any other node encodes a string in the alphabet. We indicate whether a particular string is indeed in the set of keys we seek to store by associating a bit with each alphabet symbol at each node to indicate whether that alphabet symbol at that node terminates a string we seek to store.

In our example, each node is represented as a triple, because in our example, $|\Sigma| = 3$. Each entry in the triple corresponds to one of our alphabet symbols, e.g., "a." Each entry comprises: (i) whether this alphabet symbol in this node terminates a string, and, (ii) an optional child node, if there is another string we seek to store of which the string up to this node is a prefix.

For our example of storing $\{a, ba, baa, bad, dab\}$ such a Trie can be visualized as follows.



In the root node, the entry that corresponds to the symbol "a" has associated bit 1 because "a" is in the set of strings we seek to store. However, no other string we seek to store has "a" as a prefix. Therefore, there is no subtree that starts as a child of "a" in the root node. The prefix strings "d," and "da" have subtrees associated with them because each is a prefix of some string that is in the set, which, in our example, is the string "dab."

Our python3 encoding Instead of using pointers as the above figure suggests, we use nested lists. That is, if our alphabet is $\langle a, b, d \rangle$, then, each node is of the form [[•, True or False], [•, True or False], [•, True or False]], where each • is a list, which is possibly empty, i.e., []. Thus, the set of strings from our example above would be encoded as the python3 list:

[[[], True], [[[[[[], True], [[], False], [[], True]], True], [[], False], [[], False]], False], [[[[[[], False], [[], True], [[], False]], False], [[], False], [[], False]], False]].

(You can copy-n-paste that list into your python3 interpreter to examine and manipulate.)

For example, if T is the above list which encodes our Trie, $T[0]$ is [[], True], thereby indicating that the string "a" is in our set, but "a" is not a prefix for any (other) string in the set. As another example, $T[2][0][0][1]$ is False because "da" is not in our set. $T[2][0][0][0]$, however, is not the empty list [], which indicates that there is some string in our set with prefix "da."

Your task We adopt the English alphabet, i.e., $\Sigma = \{a, \ldots, z\}$. We already provide you a subroutine SearchInTrie, which searches for a string in that alphabet in a given Trie. You need to devise and code two subroutines: InsertIntoTrie and DeleteFromTrie. The Trie you build must be "structurally sound." Your Trie is deemed to be structurally sound if and only if our SearchInTrie works correctly against your Trie, and your InsertIntoTrie and DeleteFromTrie work complementarily to one another. For example, if we insert three strings starting with the empty Trie $T = []$, and then delete those three strings, we should be left with $T = []$.