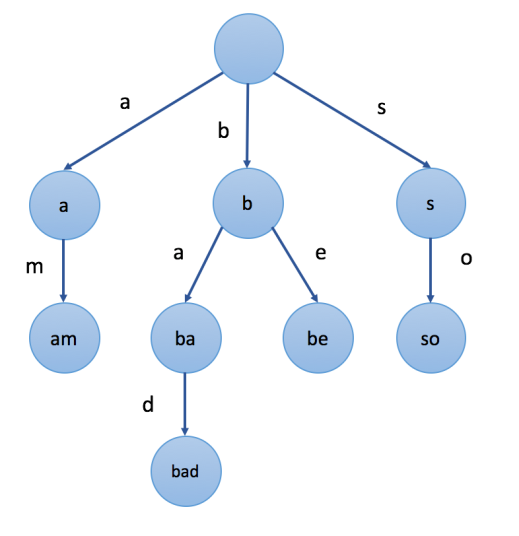
What is Trie?

* A Trie is a special form of a Nary tree.
* Typically, a trie is used to store strings.
* It is worth noting that the root node is associated with the empty string.
* One important property of Trie is that all the descendants of a node have a common prefix of the string associated with that node. That's why Trie is also called prefix tree.

Since this is a fairly advanced topic, please watch these resources before coming to the meetup  
Data Structures: Tries <https://www.youtube.com/watch?v=zIjfhVPRZCg>  
Insert Search and delete from a trie <https://www.youtube.com/watch?v=AXjmTQ8LEoI>  
Data Structures: Solve 'Contacts' Using Tries <https://www.youtube.com/watch?v=vlYZb68kAY0>

Each Trie node represents a string (a prefix). Each node might have several children nodes while the paths to different children nodes represent different characters. And the strings the child nodes represent will be the origin string represented by the node itself plus the character on the path.

Here is an example of a trie:



In the example, the value we mark in each node is the string the node represents. For instance, we start from the root node and choose the second path 'b', then choose the first child 'a', and choose child 'd', finally we arrived at the node "bad". The value of the node is exactly formed by the letters in the path from the root to the node sequentially.

Let's look at the example again. For example, the strings represented by nodes in the subtree rooted at node "b" have a common prefix "b". And vice versa. The strings which have the common prefix "b" are all in the subtree rooted at node "b" while the strings with different prefixes will come to different branches.

Trie is widely used in various applications, such as autocomplete, spell checker, etc. We will introduce the practical applications in later chapters.

How to represent a Trie?

In the previous article, we introduce the concept of Trie. In this article, we will talk about how to represent this data structure in coding languages.

Briefly review the node structure of a Nary tree before reading the following contents.

What's special about Trie is the corresponding relationship between characters and children nodes. There are a lot of different representations of a trie node. Here we provide two of them.

***First Solution - Array***

The first solution is to use an array to store children nodes.

For instance, if we store strings which only contains letter a to z, we can declare an array whose size is 26 in each node to store its children nodes. And for a specific character c, we can use c - 'a' as the index to find the corresponding child node in the array.

It is really fast to visit a child node. It is comparatively easy to visit a specific child since we can easily transfer a character to an index in most cases. But not all children nodes are needed. So there might be some waste of space.

struct TrieNode {

TrieNode\* children[N];

// you might need some extra values according to different cases

};

/\*\* Usage:

\* Initialization: TrieNode root = new TrieNode();

\* Return a specific child node with char c: (root->children)[c - 'a']

\*/

***Second Solution - Map***

The second solution is to use a hashmap to store children nodes.

We can declare a hashmap in each node. The key of the hashmap are characters and the value is the corresponding child node.

Insertion in Trie

We have talked about insertion in a BST in another card ([Introduction to Data Structure - Binary Search Tree](https://leetcode.com/explore/learn/card/introduction-to-data-structure-binary-search-tree/)).

Question:

Do you remember how to insert a new node in a binary search tree?

When we insert a target value into a BST, in each node, we need to decide which child node to go according to the relationship between the value of the node and the target value. Similarly, when we insert a target value into a Trie, we will also decide which path to go depending on the target value we insert.

To be more specific, if we insert a string S into Trie, we start with the root node. We will choose a child or add a new child node depending on S[0], the first character in S. Then we go down to the second node and we will make a choice according to S[1]. Then we go down to the third node, so on and so for. Finally, we traverse all characters in S sequentially and reach the end. The end node will be the node which represents the string S.

Here is an example:

1 / 5

Let's summarize the strategy using pseudo-code:

1. Initialize: cur = root

2. for each char c in target string S:

3. if cur does not have a child c:

4. cur.children[c] = new Trie node

5. cur = cur.children[c]

6. cur is the node which represents the string S

Usually, you will need to build the trie by yourself. Building a trie is actually to call the insertion function several times. But remember to initialize a root node before you insert the strings.

Search in Trie

* [Search Prefix](https://leetcode.com/explore/learn/card/trie/147/basic-operations/1060/#search-prefix)
* [Search Word](https://leetcode.com/explore/learn/card/trie/147/basic-operations/1060/#search-word)

**Search Prefix**

As we mentioned in the introduction to Trie, all the descendants of a node have a common prefix of the string associated with that node. Therefore, it should be easy to search if there are any words in the trie that starts with the given prefix.

Similarly, we can go down the tree depending on the given prefix. Once we can not find the child node we want, search fails. Otherwise, search succeeds. To be more specific, we provide several examples:

1 / 10

Let's summarize the strategy using pseudo-code:

1. Initialize: cur = root

2. for each char c in target string S:

3. if cur does not have a child c:

4. search fails

5. cur = cur.children[c]

6. search successes

**Search Word**

You might also want to know how to search for a specific word rather than a prefix. We can treat this word as a prefix and search in the same way we mentioned above.

1. If search fails which means that no words start with the target word, the target word is definitely not in the Trie.
2. If search succeeds, we need to check if the target word is only a prefix of words in Trie or it is exactly a word. To solve this problem, you might want to modify the node structure a little bit.

Hint: A boolean flag in each node might work.