***Trie***

*Trie data structure is defined as a Tree based data structure that is used for storing some collection of strings and performing efficient search operations on them. The word Trie is derived from re****TRIE****val, which means finding something or obtaining it.*

*Trie follows some property that If two strings have a common prefix then they will have the same ancestor in the trie. A trie can be used to sort a collection of strings alphabetically as well as search whether a string with a given prefix is present in the trie or not.*

**Need for Trie Data Structure?**

A Trie data structure is used for storing and retrieval of data and the same operations could be done using another data structure which is Hash Table but Trie can perform these operations more efficiently than a Hash Table. Moreover, Trie has its own advantage over the Hash table. A Trie data structure can be used for **prefix-based** searching whereas a Hash table can’t be used in the same way.

**Advantages of Trie Data Structure over a Hash Table:**

The A trie data structure has the following advantages over a hash table:

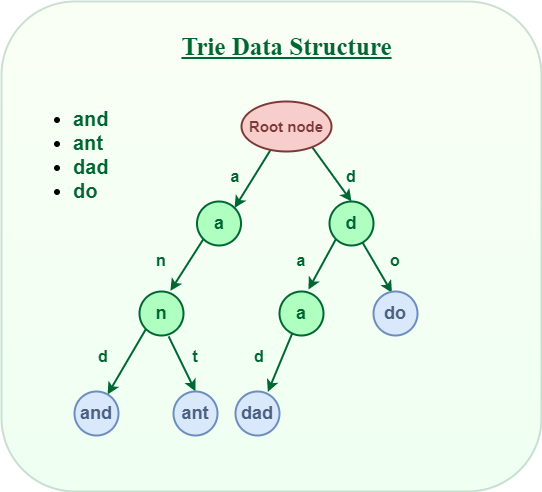
* We can efficiently do **prefix search** (or auto-complete) with Trie.
* We can easily print all words in alphabetical order which is not easily possible with hashing.
* There is no overhead of Hash functions in a Trie data structure.
* Searching for a String even in the large collection of strings in a Trie data structure can be done in **O(L)** Time complexity, Where L is the number of words in the query string. This searching time could be even less than O(L) if the query string does not exist in the trie.

**Properties of a Trie Data Structure**

Now we already know that Trie has a tree-like structure. So, it is very important to know its properties.  
Below are some important properties of the Trie data structure:

* There is one root node in each Trie.
* Each node of a Trie represents a string and each edge represents a character.
* Every node consists of hashmapsor**an array of pointers**, with each index representing a character and a flag to indicate if any string ends at the current node.
* Trie data structure can contain any number of characters including **alphabets**, **numbers**, and **special characters**. But for this article, we will discuss strings with characters a-z. Therefore, only 26 pointers need for every node, where the**0th** index represents**‘a’** and the **25th** index represents**‘z’** characters.
* Each path from the root to any node represents a word or string.

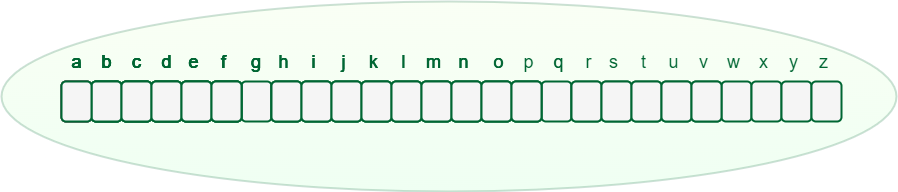
Below is a simple example of Trie data structure.



**How does Trie Data Structure work?**

We already know that the Trie data structure can contain any number of characters including **alphabets**, **numbers**, and **special characters**. But for this article, we will discuss strings with characters **a-z**. Therefore, only 26 pointers need for every node, where the**0th** index represents**‘a’** and the **25th** index represents**‘z’** characters.

Any lowercase English word can start with**a-z**, then the next letter of the word could be **a-z,** the third letter of the word again could be **a-z**, and so on. So for storing a word, we need to take an array (container) of size **26**and initially, all the characters are empty as there are no words and it will look as shown below.

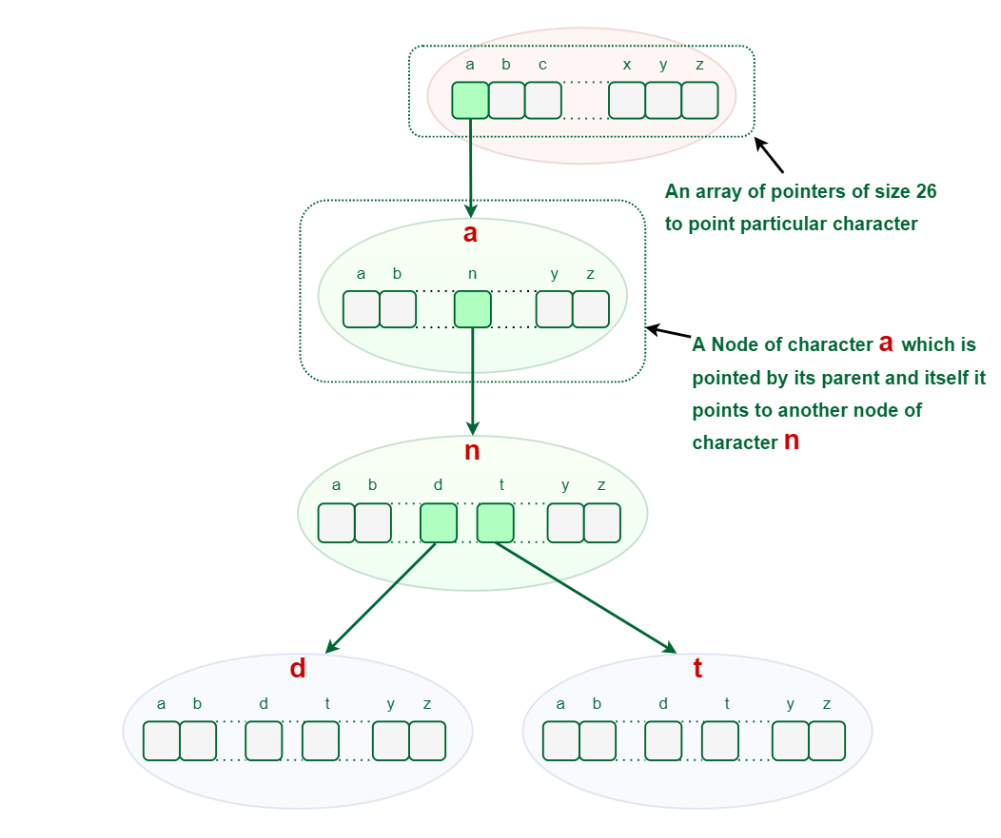


*An array of pointers inside every Trie node*

Let’s see how a word “**and**” and “**ant**” is stored in the Trie data structure:

1. Store “**and**” in Trie data structure:
   * The word “**and**” starts with “**a**“, So we will mark the position “**a**” as filled in the Trie node, which represents the use of “a”.
   * After placing the first character, for the second character again there are **26 possibilities**, So from “**a**“, again there is an array of size **26**, for storing the 2nd character.
   * The second character is “**n**“, So from “**a**“, we will move to “**n**” and mark “**n**” in the **2nd**array as used.
   * After “**n**“, the 3rd character is “**d**“, So mark the position “**d**” as used in the respective array.
2. Store “**ant**” in the Trie data structure:
   * The word “**ant**” starts with “**a**” and the position of “**a**” in the root node has already been filled. So, no need to fill it again, just move to the node ‘**a**‘ in Trie.
   * For the second character ‘**n**‘ we can observe that the position of ‘n’ in the ‘a’ node has already been filled. So, no need to fill it again, just move to node ‘n’ in Trie.
   * For the last character ‘**t**‘ of the word, The position for ‘**t**‘ in the ‘**n**‘ node is not filled. So, filled the position of ‘**t**‘ in ‘**n**‘ node and move to ‘**t**‘ node.

After storing the word “and” and “ant” the Trie will look like this:



## Representation of Trie Node:

Every Trie node consists of a character pointer array or hashmap and a flag to represent if the word is ending at that node or not. But if the words contain only **lower-case** letters (i.e. a-z), then we can define Trie Node with an array instead of a hashmap.