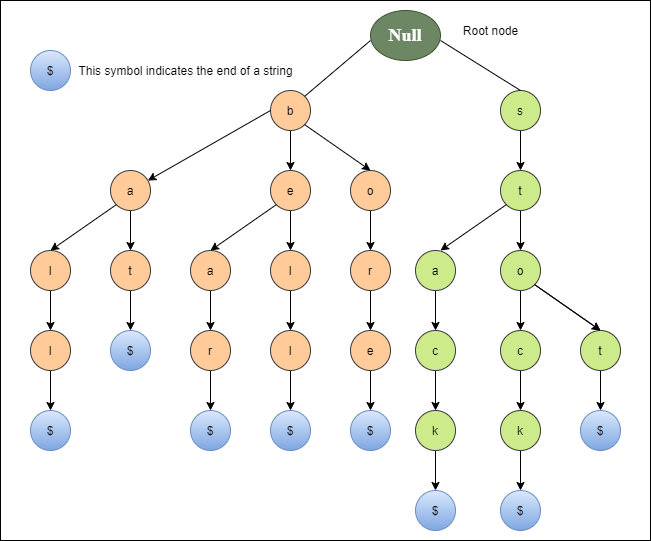
The word "**Trie**" is an excerpt from the word "**retrieval**". Trie is a sorted tree-based data-structure that stores the set of strings. It has the number of pointers equal to the number of characters of the alphabet in each node. It can search a word in the dictionary with the help of the word's prefix. For example, if we assume that all strings are formed from the letters '**a**' to '**z**' in the English alphabet, each trie node can have a maximum of **26** points.

Trie is also known as the digital tree or prefix tree. The position of a node in the Trie determines the key with which that node is connected.

## Properties of the Trie for a set of the string:

1. The root node of the trie always represents the null node.
2. Each child of nodes is sorted alphabetically.
3. Each node can have a maximum of **26** children (A to Z).
4. Each node (except the root) can store one letter of the alphabet.

The diagram below depicts a trie representation for the bell, bear, bore, bat, ball, stop, stock, and stack.



## Basic operations of Trie

There are three operations in the Trie:

1. Insertion of a node
2. Searching a node
3. Deletion of a node

### Insert of a node in the Trie

The first operation is to insert a new node into the trie. Before we start the implementation, it is important to understand some points:

1. Every letter of the input key (word) is inserted as an individual in the Trie\_node. Note that children point to the next level of Trie nodes.
2. The key character array acts as an index of children.
3. If the present node already has a reference to the present letter, set the present node to that referenced node. Otherwise, create a new node, set the letter to be equal to the present letter, and even start the present node with this new node.
4. The character length determines the depth of the trie.

**Implementation of insert a new node in the Trie**

1. **public** **class** Data\_Trie {
2. **private** Node\_Trie root;
3. **public** Data\_Trie(){
4. **this**.root = **new** Node\_Trie();
5. }
6. **public** **void** insert(String word){
7. Node\_Trie current = root;
8. **int** length = word.length();
9. **for** (**int** x = 0; x < length; x++){
10. **char** L = word.charAt(x);
11. Node\_Trie node = current.getNode().get(L);
12. **if** (node == **null**){
13. node = **new** Node\_Trie ();
14. current.getNode().put(L, node);
15. }
16. current = node;
17. }
18. current.setWord(**true**);
19. }
20. }

### Searching a node in Trie

The second operation is to search for a node in a Trie. The searching operation is similar to the insertion operation. The search operation is used to search a key in the trie. The implementation of the searching operation is shown below.

Implementation of search a node in the Trie

1. **class** Search\_Trie {
3. **private** Node\_Trie Prefix\_Search(String W) {
4. Node\_Trie node = R;
5. **for** (**int** x = 0; x < W.length(); x++) {
6. **char** curLetter = W.charAt(x);
7. **if** (node.containsKey(curLetter))
8. {
9. node = node.get(curLetter);
10. }
11. **else** {
12. **return** **null**;
13. }
14. }
15. **return** node;
16. }
18. **public** **boolean** search(String W) {
19. Node\_Trie node = Prefix\_Search(W);
20. **return** node != **null** && node.isEnd();
21. }
22. }

### Deletion of a node in the Trie

The Third operation is the deletion of a node in the Trie. Before we begin the implementation, it is important to understand some points:

1. If the key is not found in the trie, the delete operation will stop and exit it.
2. If the key is found in the trie, delete it from the trie.

**Implementation of delete a node in the Trie**

1. **public** **void** Node\_delete(String W)
2. {
3. Node\_delete(R, W, 0);
4. }
6. **private** **boolean** Node\_delete(Node\_Trie current, String W, **int** Node\_index) {
7. **if** (Node\_index == W.length()) {
8. **if** (!current.isEndOfWord()) {
9. **return** **false**;
10. }
11. current.setEndOfWord(**false**);
12. **return** current.getChildren().isEmpty();
13. }
14. **char** A = W.charAt(Node\_index);
15. Node\_Trie node = current.getChildren().get(A);
16. **if** (node == **null**) {
17. **return** **false**;
18. }
19. **boolean** Current\_Node\_Delete = Node\_delete(node, W, Node\_index + 1) && !node.isEndOfWord();
21. **if** (Current\_Node\_Delete) {
22. current.getChildren().remove(A);
23. **return** current.getChildren().isEmpty();
24. }
25. **return** **false**;
26. }

## Applications of Trie

**1. Spell Checker**

Spell checking is a three-step process. First, look for that word in a dictionary, generate possible suggestions, and then sort the suggestion words with the desired word at the top.

Trie is used to store the word in dictionaries. The spell checker can easily be applied in the most efficient way by searching for words on a data structure. Using trie not only makes it easy to see the word in the dictionary, but it is also simple to build an algorithm to include a collection of relevant words or suggestions.

**2. Auto-complete**

Auto-complete functionality is widely used on text editors, mobile applications, and the Internet. It provides a simple way to find an alternative word to complete the word for the following reasons.

* It provides an alphabetical filter of entries by the key of the node.
* We trace pointers only to get the node that represents the string entered by the user.
* As soon as you start typing, it tries to complete your input.

**3. Browser history**

It is also used to complete the URL in the browser. The browser keeps a history of the URLs of the websites you've visited.

## Advantages of Trie

1. It can be insert faster and search the string than hash tables and binary search trees.
2. It provides an alphabetical filter of entries by the key of the node.

## Disadvantages of Trie

1. It requires more memory to store the strings.
2. It is slower than the hash table.