Trie

The trie data structure, also known as prefix tree, is a tree-like data structure used for efficient retrieval of key-value pairs. It is commonly used for implementing :-

1. dictionaries
2. auto complete features

making it a fundamental component in many search algorithms.

Auto-complete = prefix search

What is Trie Data Structure?

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

Trie data structure follows a property that If two strings have a common prefix then they will have the same ancestor in the trie. This particular property allows to find all words with a given prefix.

What is need of 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 data structure 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 and a sorted traversal of all words. So a Trie has advantages of both hash table and self balancing binary search trees. However the main issue with Trie is extra memory space required to store words and the space may become huge for long list of words and/or for long words.

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 :-

- Each trie has an empty root node, with links (or references) to other nodes.

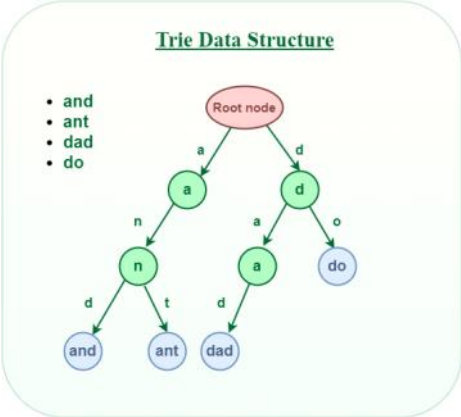
- Each node ... (or references) to other nodes.

- Each node of a trie represents a string & each edge represents a character.

  - Every node consists of hashmaps or an array of pointers, with each index representing a character & a flag to indicate if any string ends at the current node.

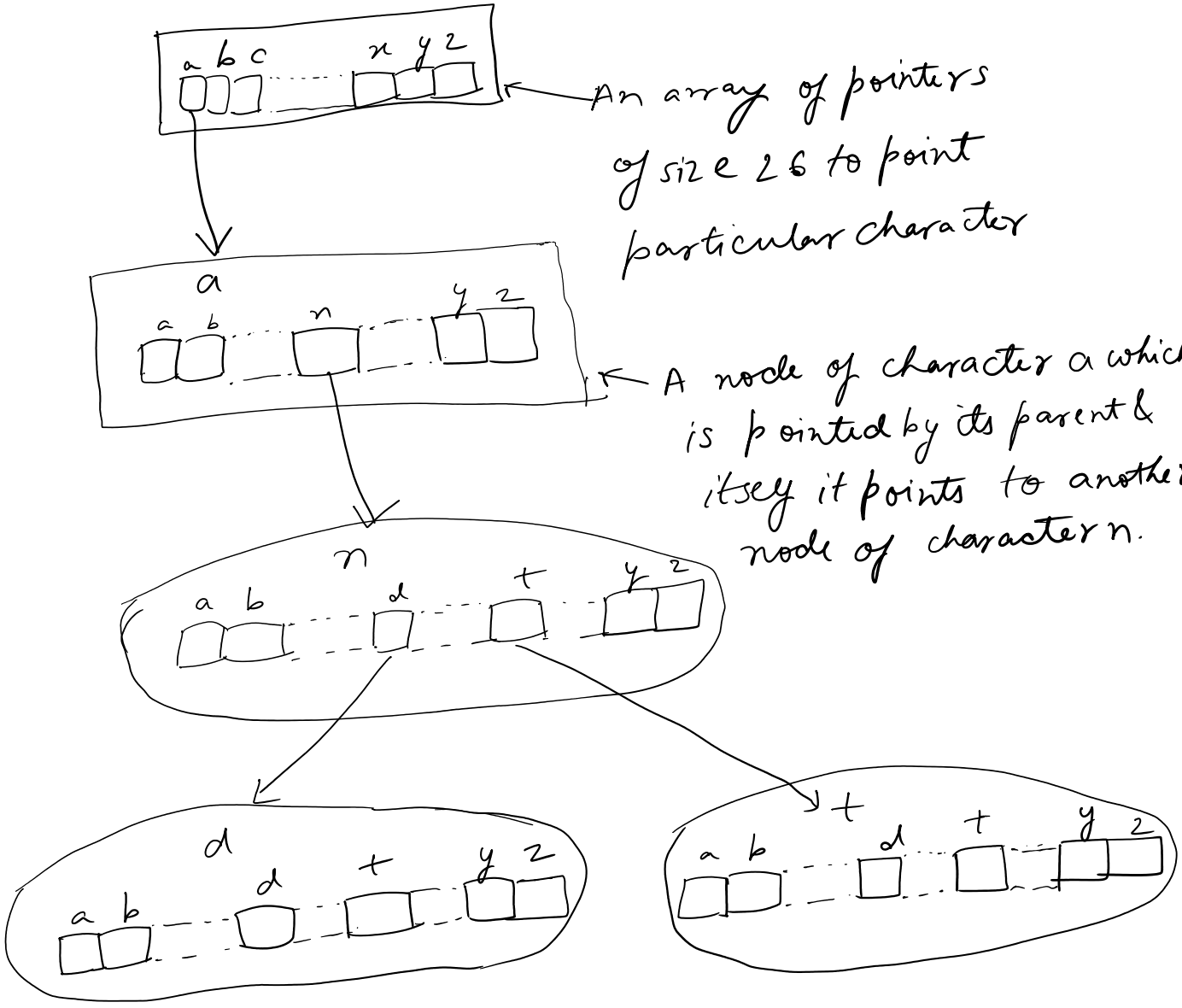
  - Trie data structure can contain any number of characters including alphabets, numbers & special characters. But for this article we will discuss strings with characters a-z. Therefore, only 26 pointers are needed for every node, where the 0th index represents 'a' & the 25th index represents 'z' characters.

  - Each path from the root to any node represents a word or string.



How does Trie Data Structure work?

eg. Below is a trie of words ["and", "ant"]



Representation of Trie Node :-

We will use a character pointer array or

Representation of Trie Node :-

Every Trie Node consists of a character pointer array or hashmap & a flag to represent if the word is ending at that or node. 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.

```
struct TrieNode {
    struct TrieNode* children[ALPHABET_SIZE];

// This will keep track of number of strings that are
// stored in the Trie from root node to any Trie
//node

int wordCount = 0;
};
```

Basic Operations on Trie Data Structure :-

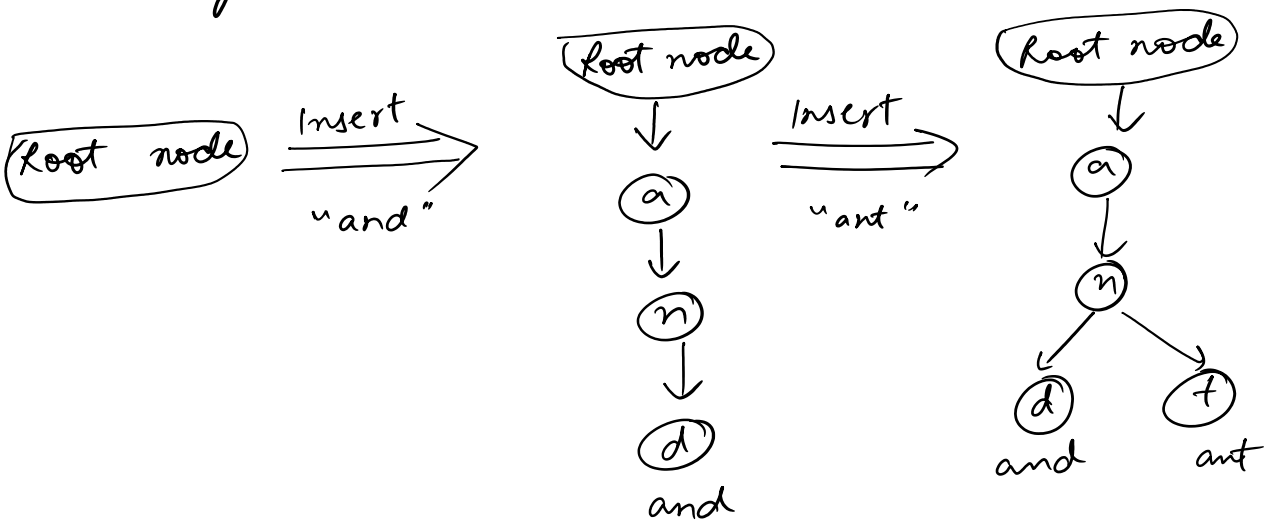
1. Insertion
2. Search
3. Deletion

1. Insertion in Trie Data Structure :-

Used to insert new strings into the Trie Data Structure.

Let us try to insert "and" & "ant" in this trie.

Let us try to insert "and" & "ant" in this trie.



From the above representation of insertion, we can see that the word "and" & "ant" have shared some common node (i.e "an") this is because of the property of Trie data structure that if two strings have a common prefix then they will have the same ancestor in the trie.

Implementation of Insertion in Trie data structure:

Algorithm:

1. Define a function **insert(TrieNode *root, string &word)** which will take two parameters one for the root and the other for the string that we want to insert in the Trie data structure.
2. Now take another pointer **currentNode** and initialize it with the **root** node.
3. Iterate over the length of the given string and check if the value is **NULL** or not in the array of pointers at the current character of the string.
   - If It's **NULL** then, make a new node and point the current character to this newly created node.
   - Move the curr to the newly created node.
4. Finally, increment the **wordCount** of the last **currentNode,** this implies that there is a string ending currentNode.

```
void insert_key(TrieNode* root, string& key){
//Initialize the currentNode pointer with the root node
TrieNode* currentNode = root;
// Iterate across the length of the string
```

```
// Iterate across the length of the string
for (auto c : key){
  // Check if the node exist for the current
  // character in the Trie
if (currentNode -> childNode[c - 'a'] == NULL){
  // If node for current character does not exist
  // then make a new node
  TrieNode* newNode = new TrieNode();
    // Keep the reference for the newly created node
  currentNode -> childNode[c - 'a'] = newNode;
  }
  // Now move the currentNode pointer to the
  // newly created node
  currentNode = currentNode -> childNode[c - 'a'];
  }
  //Increment the wordEndCount for the last currentNode
  // pointer this implies that there is a string ending at
  //currentNode
  currentNode -> wordCount++;
  }
```

Searching in Trie Data Structure:-

Search operation in Trie is performed in a similar way as the insertion operation but the only difference is that whenever we find that the array of pointers in **curr node** does not point to the **current character** of the **word** then return false instead of creating a new node for that current character of the word.

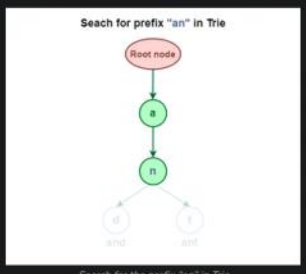
This operation is used to search whether a string is present in the Trie data structure or not. There are two search approaches in the Trie data structure.

1. Find whether the given word exists in Trie.
2. Find whether any word that starts with the given prefix exists in Trie.

There is a similar search pattern in both approaches. The first step in searching a given word in Trie is to convert the word to characters and then compare every character with the trie node from the root node. If the current character is present in the node, move forward to its children. Repeat this process until all characters are found.

2.1 Searching Prefix in Trie Data Structure:

Search for the prefix "**an**" in the Trie Data Structure.



Search for the prefix "an" in Trie

Implementation of Prefix Search in Trie Data Structure :-

```
bool isPrefixExist(TrieNode* root, string &key){
//Initialize the currentNode pointer
//with the root node
TrieNode* currentNode = root;
//Iterate across the length of the string
for(auto c:key){
  //check if the node exist for the current
  //character in the Trie
  if(currentNode->childNode[c-'a']==NULL){
  //Given word as a prefix does not exist in Trie
    return false);}
    // Move the currentNode pointer to the already
    //existing Node for current character
```

// existing Node for current character
currentNode = currentNode -> childNode[c - 'a'];
}
// Prefix exist in the Trie
return true; }

Searching Complete word in Trie Data Structure :-

C++ Java Python C# JavaScript

```cpp
bool search_key(TrieNode* root, string& key)
{
    // Initialize the currentNode pointer
    // with the root node
    TrieNode* currentNode = root;

    // Iterate across the length of the string
    for (auto c : key) {

        // Check if the node exist for the current
        // character in the Trie.
        if (currentNode->childNode[c - 'a'] == NULL) {

            // Given word does not exist in Trie
            return false;
        }

        // Move the currentNode pointer to the already
        // existing node for current character.
        currentNode = currentNode->childNode[c - 'a'];
    }

    return (currentNode->wordCount > 0);
}
```

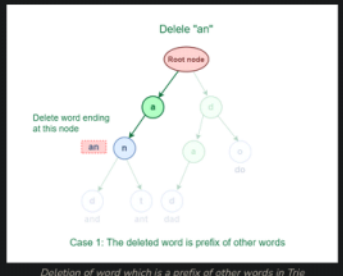
3. Deletion in Trie Data Structure

This operation is used to delete strings from the Trie data structure. There are three cases when deleting a word from Trie

1. The deleted word is a prefix of other words in Trie.
2. The deleted word shares a common prefix with other words in Trie.
3. The deleted word does not share any common prefix with other words in Trie.

3.1 The deleted word is a prefix of other words in Trie.

As shown in the following figure, the deleted word "**an**" share a complete prefix with another word "**and**" and "**ant**".



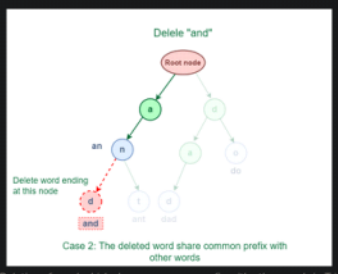
Deletion of word which is a prefix of other words in Trie

An easy solution to perform a delete operation for this case is to just decrement the **wordCount** by 1 at the ending node o the word.

3.2 The deleted word shares a common prefix with other words in Trie.

As shown in the following figure, the deleted word "**and**" has some common prefixes with other words '**ant**'. They share the prefix '**an**'.

As shown in the following figure, the deleted word "**and**" has some common prefixes with other words '**ant**'. They share the prefix '**an**'.

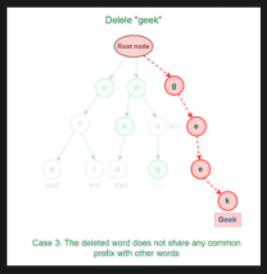


Deletion of word which shares a common prefix with other words in Trie

The solution for this case is to delete all the nodes starting from the end of the prefix to the last character of the given word.

3.3 The deleted word does not share any common prefix with other words in Trie.

As shown in the following figure, the word "**geek**" does not share any common prefix with any other words.



The solution for this case is just to delete all the nodes.

Below is the implementation that handles all the above cases:

C++ Java Python C# JavaScript

```
bool delete_key(TrieNode* root, string& word)
{
    TrieNode* currentNode = root;
    TrieNode* lastBranchNode = NULL;
    char lastBranchChar = 'a';

    for (auto c : word) {
        if (currentNode->childNode[c - 'a'] == NULL) {
            return false;
        }
        else {
            int count = 0;
            for (int i = 0; i < 26; i++) {
```



```
            }

            if (count > 1) {
                lastBranchNode = currentNode;
                lastBranchChar = c;
            }
            currentNode = currentNode->childNode[c - 'a'];
        }
    }

    int count = 0;
    for (int i = 0; i < 26; i++) {
        if (currentNode->childNode[i] != NULL)
            count++;
    }

    // Case 1: The deleted word is a prefix of other words
    // in Trie.
    if (count > 0) {
        currentNode->wordCount--;
        return true;
    }

    // Case 2: The deleted word shares a common prefix with
    // other words in Trie.
    if (lastBranchNode != NULL) {
        lastBranchNode->childNode[lastBranchChar] = NULL;
        return true;
    }
    // Case 3: The deleted word does not share any common
    // prefix with other words in Trie.
    else {
        root->childNode[word[0]] = NULL;
        return true;
    }
}
```

Implement Trie Data Structure?

Algorithm:

- Create a root node with the help of **TrieNode()** constructor.
- Store a collection of strings that we have to insert in the trie in a vector of strings say, **arr**.
- Inserting all strings in Trie with the help of the **insertkey()** function,
- Search strings from **searchQueryStrings** with the help of **search_key()** function.
- Delete the strings present in the **deleteQueryStrings** with the help of **delete_key**.

C++ Python C# JavaScript

```
#include <bits/stdc++.h>
```

C++ Python C# JavaScript

```cpp
#include <bits/stdc++.h>
using namespace std;

struct TrieNode {

    // pointer array for child nodes of each node
    TrieNode* childNode[26];
    int wordCount;

    TrieNode()
    {
        // constructor
        // initialize the wordCnt variable with 0
        // initialize every index of childNode array with
        // NULL
        wordCount = 0;
        for (int i = 0; i < 26; i++) {
            childNode[i] = NULL;
        }
    }
};

void insert_key(TrieNode* root, string& key)
{
    // Initialize the currentNode pointer
    // with the root node
    TrieNode* currentNode = root;

    // Iterate across the length of the string
    for (auto c : key) {

        // Check if the node exist for the current
        // character in the Trie.
        if (currentNode->childNode[c - 'a'] == NULL) {

            // If node for current character does not exist
            // then make a new node
            TrieNode* newNode = new TrieNode();

            // Keep the reference for the newly created
            // node.
            currentNode->childNode[c - 'a'] = newNode;
        }

        // Now, move the current node pointer to the newly
        // created node.
        currentNode = currentNode->childNode[c - 'a'];
    }

    // Increment the wordEndCount for the last currentNode
    // pointer this implies that there is a string ending at
    // currentNode.
    currentNode->wordCount++;
}

bool search_key(TrieNode* root, string& key)
{
    // Initialize the currentNode pointer
    // with the root node
    TrieNode* currentNode = root;

    // Iterate across the length of the string
    for (auto c : key) {

        // Check if the node exist for the current
        // character in the Trie.
        if (currentNode->childNode[c - 'a'] == NULL) {

            // Given word does not exist in Trie
            return false;
        }

        // Move the currentNode pointer to the already
        // existing node for current character.
        currentNode = currentNode->childNode[c - 'a'];
    }

    return (currentNode->wordCount > 0);
}

bool delete_key(TrieNode* root, string& word)
{
    TrieNode* currentNode = root;
    TrieNode* lastBranchNode = NULL;
    char lastBrachChar = 'a';

    for (auto c : word) {
        if (currentNode->childNode[c - 'a'] == NULL) {
            return false;
        }
        else {
            int count = 0;
            for (int i = 0; i < 26; i++) {
                if (currentNode->childNode[i] != NULL)
                    count++;
            }

            if (count > 1) {
                lastBranchNode = currentNode;
                lastBrachChar = c;
            }
            currentNode = currentNode->childNode[c - 'a'];
        }
    }

    int count = 0;
    for (int i = 0; i < 26; i++) {
        if (currentNode->childNode[i] != NULL)
            count++;
    }

    // Case 1: The deleted word is a prefix of other words
    // in Trie.
    if (count > 0) {
        currentNode->wordCount--;
        return true;
    }

    // Case 2: The deleted word shares a common prefix with
    // other words in Trie.
    if (lastBranchNode != NULL) {
        lastBranchNode->childNode[lastBrachChar] = NULL;
        return true;
    }
    // Case 3: The deleted word does not share any common
    // prefix with other words in Trie.
    else {
        root->childNode[word[0]] = NULL;
        return true;
    }
}
```

```cpp
// Driver code
int main()
{
    // Make a root node for the Trie
    TrieNode* root = new TrieNode();

    // Stores the strings that we want to insert in the
    // Trie
    vector<string> inputStrings
        = { "and", "ant", "do", "geek", "dad", "ball" };

    // number of insert operations in the Trie
    int n = inputStrings.size();

    for (int i = 0; i < n; i++) {
        insert_key(root, inputStrings[i]);
    }

    // Stores the strings that we want to search in the Trie
    vector<string> searchQueryStrings
        = { "do", "geek", "bat" };

    // number of search operations in the Trie
    int searchQueries = searchQueryStrings.size();

    for (int i = 0; i < searchQueries; i++) {
        cout << "Query String: " << searchQueryStrings[i]
             << "\n";
        if (search_key(root, searchQueryStrings[i])) {
            // the queryString is present in the Trie
            cout << "The query string is present in the "
                    "Trie\n";
        }
        else {
            // the queryString is not present in the Trie
            cout << "The query string is not present in "
                    "the Trie\n";
        }
    }

    // stores the strings that we want to delete from the
    // Trie
    vector<string> deleteQueryStrings = { "geek", "tea" };

    // number of delete operations from the Trie
    int deleteQueries = deleteQueryStrings.size();

    for (int i = 0; i < deleteQueries; i++) {
        cout << "Query String: " << deleteQueryStrings[i]
             << "\n";
        if (delete_key(root, deleteQueryStrings[i])) {
            // The queryString is successfully deleted from
            // the Trie
            cout << "The query string is successfully "
                    "deleted\n";
        }
        else {
            // The query string is not present in the Trie
            cout << "The query string is not present in "
                    "the Trie\n";
        }
    }

    return 0;
}
```

Output

```
Query String: do
The query string is present in the Trie
Query String: geek
The query string is present in the Trie
Query String: bat
The query string is not present in the Trie
Query String: geek
The query string is successfully deleted
Query String: tea
The query string is not present in the Trie
```

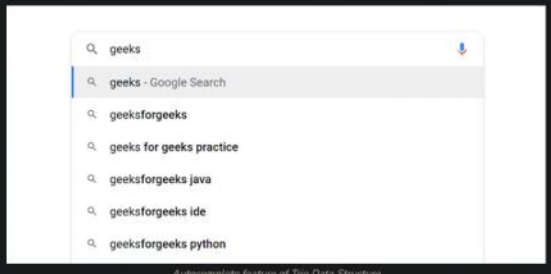
Complexity Analysis of Trie Data Structure

Operation	Time Complexity
Insertion	O(n) Here n is the length of string to be searched
Searching	O(n)
Deletion	O(n)

Note: In the above complexity table '**n**', '**m**' represents the size of the string and the number of strings that are stored in the trie.

Applications of Trie data structure:

1. Autocomplete Feature: Autocomplete provides suggestions based on what you type in the search box. Trie data structure is used to implement autocomplete functionality.



Autocomplete feature of Trie Data Structure

2. Spell Checkers: If the word typed does not appear in the dictionary, then it shows suggestions based on what you

2. Spell Checkers: If the word typed does not appear in the dictionary, then it shows suggestions based on what you typed.

It is a 3-step process that includes :

1. Checking for the word in the data dictionary.
2. Generating potential suggestions.
3. Sorting the suggestions with higher priority on top.

Trie stores the data dictionary and makes it easier to build an algorithm for searching the word from the dictionary and provides the list of valid words for the suggestion.

3. Longest Prefix Matching Algorithm(Maximum Prefix Length Match): This algorithm is used in networking by the routing devices in IP networking. Optimization of networ s contiguous masking that bound the complexity

To speed up the lookup process, Multiple Bit trie schemes were developed that perform the lookups of multiple bits faster.

Advantages of Trie data structure:

- Trie allows us to input and finds words in O(n) time, where n is the length of a single word. It is faster as compared to both hash tables and binary search trees.
- It provides alphabetical filtering of entries by the key of the node and hence makes it easier to print all words in alphabetical order.
- Prefix search/Longest prefix matching can be efficiently done with the help of trie data structure.
- Since trie doesn't need any hash function for its implementation so they are generally faster than hash tables for small keys like integers and pointers.
- Tries support ordered iteration whereas iteration in a hash table will result in pseudorandom order given by the hash function which is usually more cumbersome.
- Deletion is also a straightforward algorithm with O(n) as its time complexity, where n is the length of the word to be deleted.

Disadvantages of Trie data structure:

- The main disadvantage of the trie is that it takes a lot of memory to store all the strings. For each node, we have too many node pointers which are equal to the no of characters in the worst case.
- An efficiently constructed hash table(i.e. a good hash function and a reasonable load factor) has O(1) as lookup time which is way faster than O(l) in the case of a trie, where l is the length of the string.