# Project Report

**Member : Abhilash Kolluri(ak2048), Srikar Gona(gs943), Sachit Patel(sp2135)**

**INTRODUCTION** : TRIE also called as prefix tree is tree type data structure which is used for retrieval of a key in the dataset of strings. It's heavily used for searching a word in the huge dataset minimizing the space.
Though there are several other data structures like hash tables and balanced trees which can be used to search for a word in the dataset. TRIE stands out for below mentioned reasons. Although hash table has O(1) time complexity for looking a key, it is not efficient for

- Finding keys with a common prefix.
- Provide and correct the user suggesting words.

Coming to memory, hash tables take a considerable amount of memory. Also, with increase in the data set, not only the memory but it leads to more collisions which may lead to increase of search time complexity and defeating the purpose of using hash tables overall. This could go as bad as O(n), where n is the number of keys inserted. Trie on the other hand takes less space compared to hash tables. This is best seen when strings with the same prefix are present in our dataset. The complexity here would be O(m), where m is the length of the key. This makes its application vast in the real life world. Few examples being AutoComplete, Spell Checker, IP Routing, Word Games(boggle).

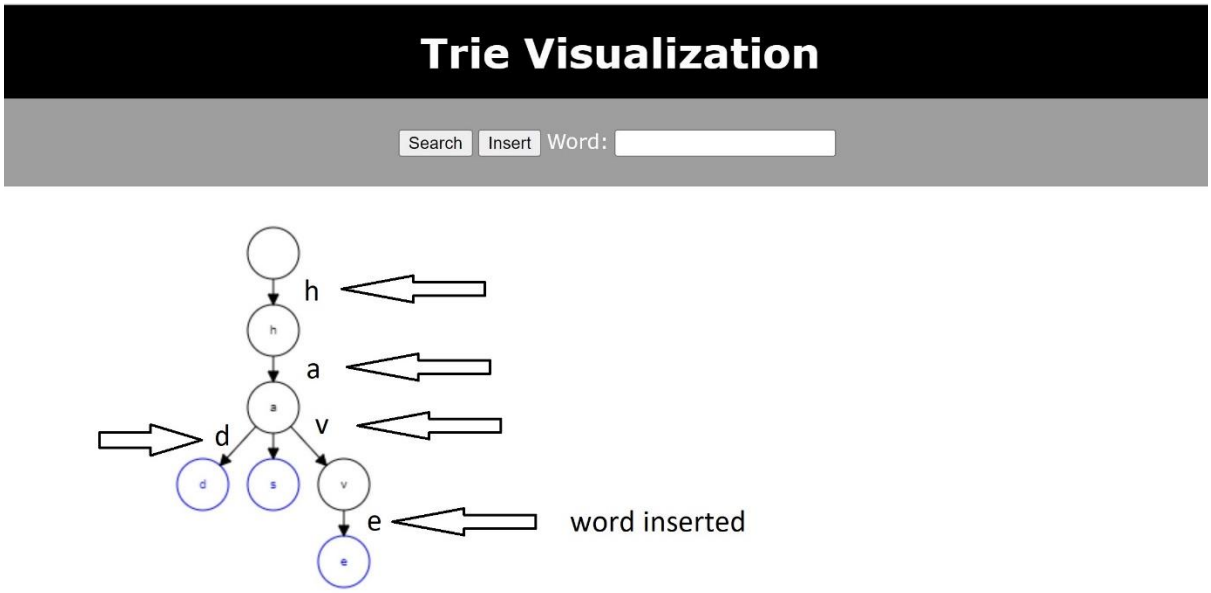
**DATA STRUCTURES USED** : Trie is a rooted tree, and initially we have a root trie structure. To implement our algorithm we have backed ourselves with a simple yet powerful trie structure. It basically has pointers to its children, Since this project we are considering only has lowercase alphabetical letters. This turns out to be 26. Hence each trie node has a link to its 26 children. Finally, a boolean variable that helps in our search to find out whether the prefix where we stopped is a word that has been initially inserted into the trie.

**ALGORITHM** : We divided the project into 3 phases while implementing. Firstly the INSERT Operation. Secondly, SEARCH operation and finally SUGGESTION Operation.

- **INSERT** : We insert a key into trie by searching the already existing trie. Initially, we have a root node. We start from the root and check if the first character of the string exists. Here 2 cases arise:

    1. If there is already a child node we move down the tree to the next child level and continue searching for the next key character.

2. If a link does not exist, then we create a new node and link it with the parent node. We keep doing this step until we encounter the last character of the key in the string that was provided as input.

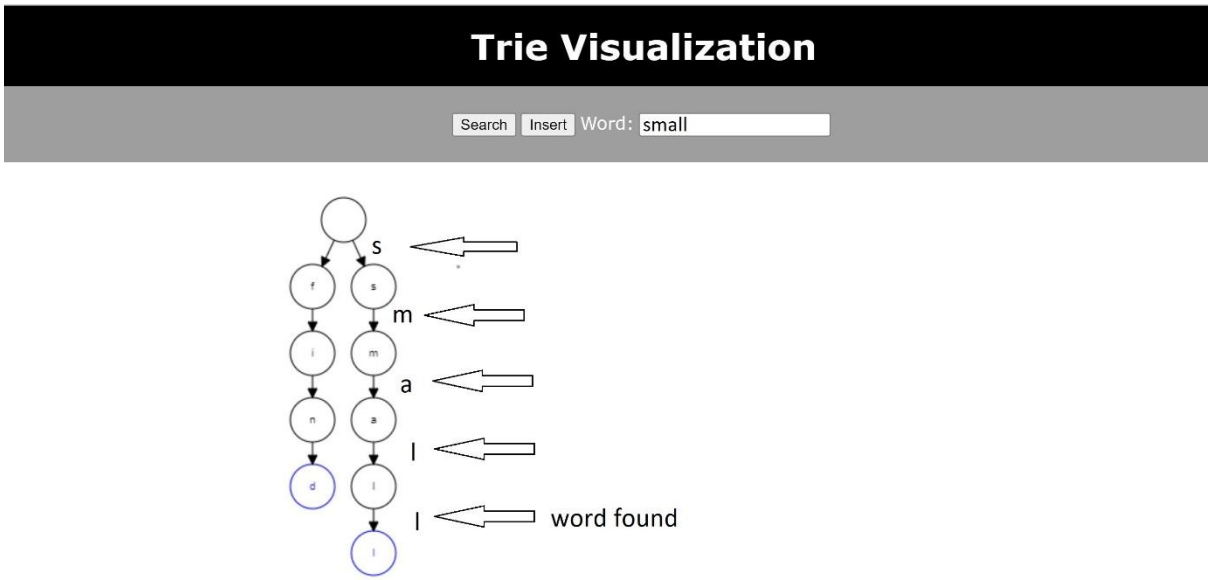



The above diagram shows how words ( has, had, have) been inserted into Trie. These words have common prefix '***ha***'

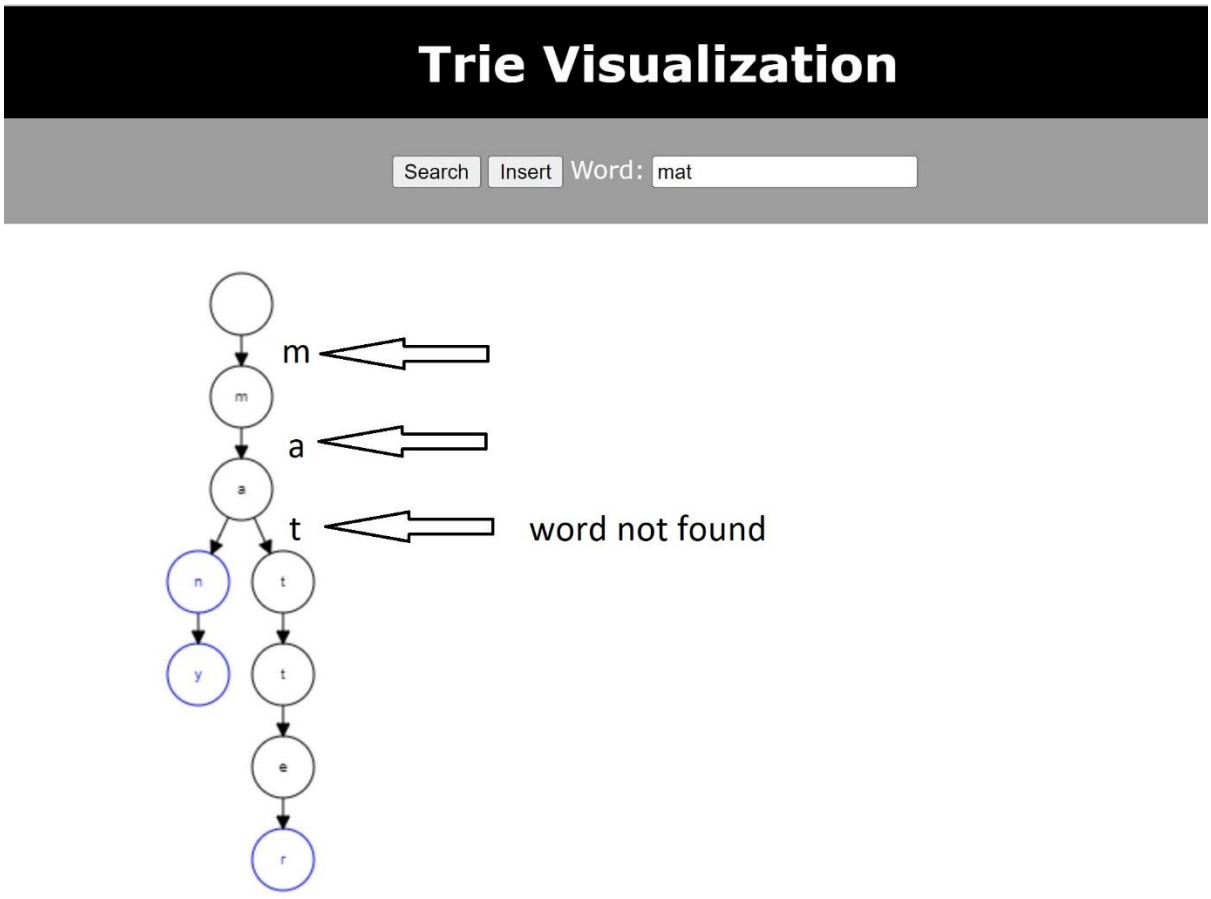
- **SEARCH** : The search algorithm is a simple follow the path approach to find the word exists or not. Each key is represented in the trie as a path starting from the root node to the leaf node or any other internal node(in case of prefix). The working is, we start from the root with the first key character and examine if a child exists that corresponds to the to the key character. Hera 2 cases arise as well :

    1. If a link exists i.e. there is a child node, we simply follow the path following the child link and search the next key character.

    2. If a child node does not exist and if there are key characters left, it is not possible to follow the path which implies the given key is not present in our dictionary app. Else, if key characters end and the last node has the boolean variable set to true, this indicates that the prefix exists in our dictionary app and it's a word too. If the boolean flag is not set, it simply implies a prefix is present in our app but it's not a valid word.




The above image is an example where a given word by user is already existing in the dictionary. It follows the path in the trie and communicates if the word is present in the dictionary or not.




The above image is an example where a given word by user does not exist in the dictionary. It follows the path in the trie and communicates if the word is present in the dictionary or not.

- **SEARCH FOR PREFIX** : The algorithm is very similar to the search for word algorithm. We traverse the trie from the root, till no key characters are found or there is no way we can continue along the path. If we are able to follow a path for the prefix provided we simply return true indicating the prefix is present in the app.

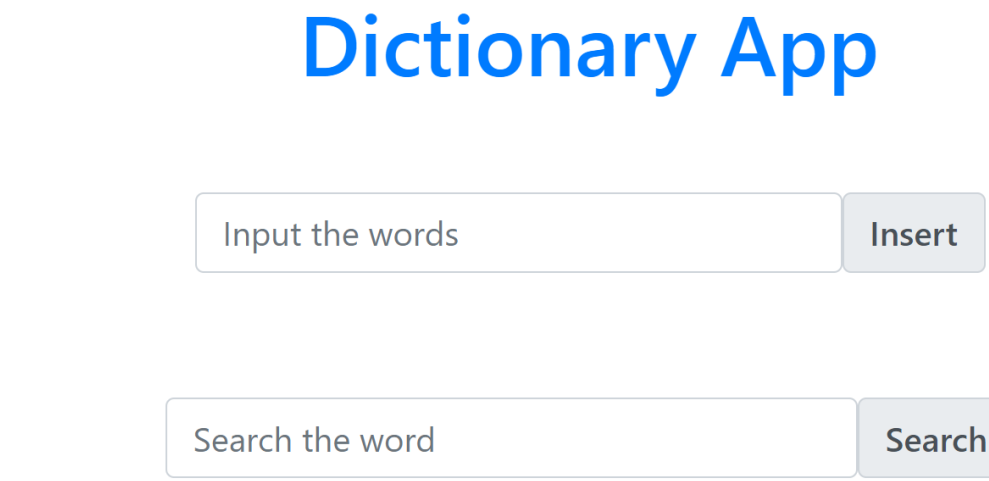
- **SUGGESTION ALGORITHM** : In the dictionary app, if we did not find the word provided different suggestions that are closest to the provided word will be displayed. We initially search the trie to see if a word is present. If a word is not present, we go to the closest prefix in the trie structure. This is achieved using the SEARCH algo. The SEARCH API then transfers the control to SUGGEST API. The SUGGEST API recursively finds all nearest words using the obtained prefix and displays all the results.

**TIME AND SPACE COMPLEXITY** :

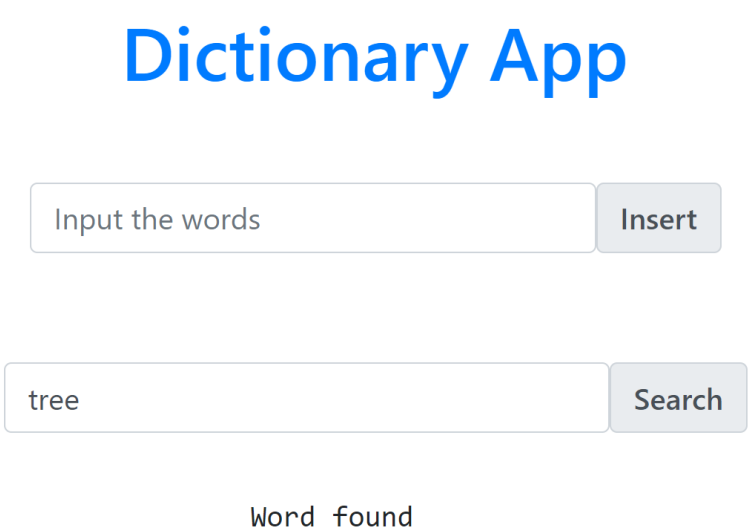
- **FOR INSERT OPERATION** : The Time complexity is O(M) where M is the word length. We travel all the key characters of the string and create a node for each if not existent, hence overall complexity is time required to scan all characters of string which is O(M). The space complexity is O(M) in the worst case, when there is no prefix existing in the trie, we would need to create M nodes.

- **FOR SEARCH OPERATION** : The Time complexity is O(M) where M is the word length. We travel all the key characters of the string to see if the word exists in the trie. This operation does not require any memory. All the required memory is already allocated as part of the INSERT operation. Hence O(1).

**WORKING of the DICTIONARY APP** :
We have imported 300 most popular English words into the trie structure. This is our initial dictionary. The user can search if the word exists in the dictionary. Below is the snapshot of the working model of the dictionary app.

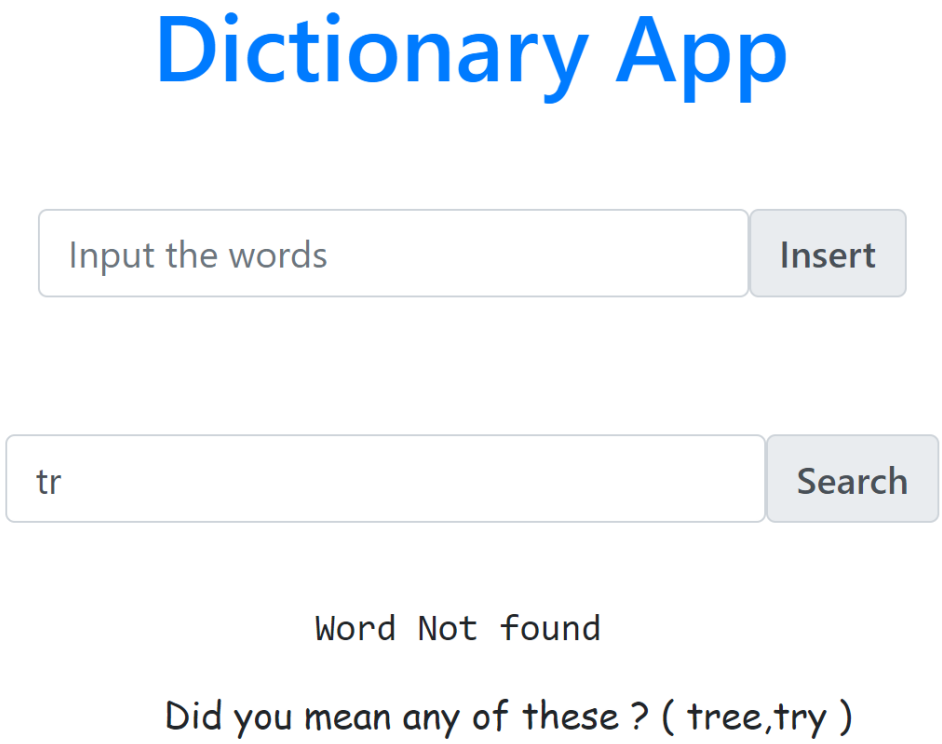



The user inserts the required word that he desires to add into the dictionary app by entering the word and clicking on the Insert button. The app will then inserts this word into the existing dictionary and marks it as a valid word.




The user can search the words in the dictionary by entering the word to be searched and clicking on the search button. If the word is present in the dictionary it displays the message as '**Word Found**'

If the word is not present, the application will throw a message saying a word not found. Along with this a bunch of suggestions that closely match with the incorrect word pops up. The user also has the option to insert words into the trie. The trie keeps building whenever an insert has been done by the user. The frontend has been implemented in Angular JS and the actual logic in JAVA Script.




**DATA STORAGE** : Our dataset is an array of strings. Initially we have taken 300 most frequent words used in english and stored them in the trie. In trie, each node represents the key character of the strings. These initial words provide a good amount of prefix for future words that are being inserted in the app by the user. The trie keeps increasing with more and more prefixes and the word count increases when the user enters different valid words.

**Technologies Used** :-

1. HTML
2. CSS
3. Type-Script
4. Angular
5. Java-Script

**High Level Description of The Project** :-

In this project we implemented many API's which include addWords ( to insert a word into the Dictionary), find ( to search the words), Suggestions ( If the searched word is not found, return some suggestions ).

1. **addWords** ( string word) :-

```
constructor() {
  this.root = new TrieNode();
}

addWord(word: string): void {

  let trieNode: TrieNode  = this.root;

  for (const letter of word) {
    if (trieNode.has(letter)) {
      trieNode = trieNode.get(letter);
    } else {
      trieNode = trieNode.setLetter(letter);
    }
  }
  // last TrieNode determines the end of a word
  trieNode.isCompleteWord = true
}
```

2. Find words:-

```
find(letters: string): string[] {
  let trieNode: TrieNode = this.root;

  trieNode.children.forEach((value: TrieNode, key: string) => {
    console.log(key, value);
  });

  for (const letter of letters) {
    trieNode = trieNode.get(letter);
    if (!trieNode) {
      return []
    }
  }

  if(trieNode.isCompleteWord == true){

    let result: Array<string>;
    result = ['temp'];
    result.pop();
    result.push(letters);
    return result;
  }

  //console.log("trienode---children"+ trieNode.children.toString());
  return trieNode.findWords(letters, [])
}
```

3. **Suggestions** ( provide word suggestions) :-

```
findWords(letters: string, results: string[]): string[] {

  this.children.forEach((trieNode, letter) => {
    const word = letters + letter;
    if (trieNode.isCompleteWord) {
      results.push(word);
    }
    trieNode.findWords(word, results);
  })
  return results;
}
```

**Steps to run the app** :-

1. Go into the Angular app folder and run ‘npm install’ to install the packages.
2. Then you can start the Project by running ‘npm start’.
3. For Animation Just open the html provided in any browser to access it.

**Note** :- The source code of the app is present in ‘my-app\src\app’ folder.

**Results** :-

Overall our app performed exceptionally well. At first Our Api’s returned true if the search words are present in the dictionary. If the Word is not found, then corresponding Message is displayed to the User and If there are any suggestions i.e., if there are any nearest words then those suggestions are also provided. End user could also insert his required words into the Dictionary.

**Additional Analysis/ Comments** :-

1. In this Project we implemented the dictionary for lower-case alphabets. We could also extend this approach to include Upper-Case/ alpha-numeric words as well.