Implementing Auto-Complete Feature using Trie

The **Auto-Complete** feature is widely useful in showing suggestions when the user types a certain word. In a simpler version, the auto-complete feature lists all of the strings which have a matching prefix queried by a user.

For example, if the dictionary stores the following words **{"abc", "abcd", "aa", "abbbaba"}** and the User types in **"ab"** then he must be shown *{"abc", "abcd", "abbbaba"}* as a result as all of them have the prefix *ab*.

Implementation using Trie

We can implement this Auto-Complete feature easily using a Trie data structure. We will have to first insert all of the strings initially in a Trie, and then fetch all matching strings based on the user's query.

Detailed Algorithm:

1. Insert all of the given strings in a Trie.
2. Search for the given query using standard Trie search algorithm.
3. If query prefix itself is not present, return -1 to indicate the same.
4. If the query is present and is the end of the word in Trie, print query. This can quickly be checked by seeing if the last matching node has the *isEndWord* flag set. We use this flag in Trie to mark the end of word nodes for the purpose of searching.
5. If the last matching node of the query has no children, return.
6. Else recursively print all nodes under a subtree of last matching node.

Dash
All
Articles
Videos
Problems
Quiz

Article marked as read.

Implementation:

C++ Java

```
// Java Program to implement Auto-Complete
// Feature using Trie

import java.util.*;
import java.io.*;
import java.lang.*;

class GFG {

    // Alphabet size (# of symbols)
    public static final int ALPHABET_SIZE = 26;

    // Trie node
    static class TrieNode
    {
        TrieNode children[] = new TrieNode[ALPHABET_SIZE];

        // isWordEnd is true if the node represents
        // end of a word
        boolean isWordEnd;
    };

    // Returns new trie node (initialized to NULLs)
    static TrieNode getNode() {
        TrieNode pNode = new TrieNode();
        pNode.isWordEnd = false;

        for(int i = 0; i < ALPHABET_SIZE; i++)
            pNode.children[i] = null;
```

Dash
All
Articles
Videos
Problems
Quiz

Article marked as read.

```
        return pNode;
    }

    // If not present, inserts key into trie. If the
    // key is prefix of trie node, just marks leaf node
    static void insert(TrieNode root, final String key)
    {
        TrieNode pCrawl = root;

        for(int level = 0; level < key.length(); level++)
        {
            int index = (key.charAt(level) - 'a');
            if (pCrawl.children[index] == null)
            pCrawl.children[index] = getNode();
            pCrawl = pCrawl.children[index];
        }

        // mark last node as leaf
        pCrawl.isWordEnd = true;
    }

    // Returns true if key presents in trie, else false
    boolean search(TrieNode root, final String key)
    {
        int length = key.length();
        TrieNode pCrawl = root;

        for (int level = 0; level < length; level++)
        {
            int index = (key.charAt(level) - 'a');

            if (pCrawl.children[index] == null)
                pCrawl = pCrawl.children[index];
        }

        return (pCrawl != null && pCrawl.isWordEnd);
    }

    // Returns 0 if current node has a child
    // If all children are NULL, return 1.
    static boolean isLastNode(TrieNode root)
```

Dash
All
Articles
Videos
Problems
Quiz

Article marked as read.

```
{
    for (int i = 0; i < ALPHABET_SIZE; i++)
        if (root.children[i] != null)
            return false;
    return true;
}

// Recursive function to print auto-suggestions
// for given node.
static void suggestionsRec(TrieNode root, String currPrefix)
{
    // found a string in Trie with the given prefix
    if (root.isWordEnd)
    {
        System.out.println(currPrefix);
    }

    // All children struct node pointers are NULL
    if (isLastNode(root))
        return;

    for (int i = 0; i < ALPHABET_SIZE; i++)
    {
        if (root.children[i] != null)
        {
            // append current character to currPrefix string
            currPrefix += (char)(97 + i);

            // recur over the rest
            suggestionsRec(root.children[i], currPrefix);
        }
    }
}

// Fucntion  to print suggestions for
// given query prefix.
static int printAutoSuggestions(TrieNode root,
                                final String query)
{
    TrieNode pCrawl = root;
```

```
// Check if prefix is present and find the
// the node (of last level) with last character
// of given string.
int level;
int n = query.length();

for (level = 0; level < n; level++)
{
    int index = (query.charAt(level) - 'a');

    // no string in the Trie has this prefix
    if (pCrawl.children[index] == null)
        return 0;

    pCrawl = pCrawl.children[index];
}

// If prefix is present as a word.
boolean isWord = (pCrawl.isWordEnd == true);

// If prefix is last node of tree (has no
// children)
boolean isLast = isLastNode(pCrawl);

// If prefix is present as a word, but
// there is no subtree below the last
// matching node.
if (isWord && isLast)
{
    System.out.println(query);
    return -1;
}

// If there are are nodes below last
// matching character.
if (!isLast)
{
    String prefix = query;
    suggestionsRec(pCrawl, prefix);
    return 1;
}
```

```
        return 0;
    }

    // Driver code
    public static void main(String[] args)
    {
        TrieNode root = getNode();
        insert(root, "hello");
        insert(root, "dog");
        insert(root, "hell");
        insert(root, "cat");
        insert(root, "a");
        insert(root, "helps");
        insert(root, "helping");
        int comp = printAutoSuggestions(root, "hel");

        if (comp == -1)
            System.out.println("No other strings found "+
                                "with this prefix\n");
        else if (comp == 0)
            System.out.println("No string found with"+
                                " this prefix\n");
    }
}
```

Output

```
hel
hell
hello
hellp
```

```
hellping
hellpis
```

Time Complexity: O(N*L) where N is the number of words in the trie and L is the

Auxiliary Space: O(N*L+N * ALPHABET_SIZE)