Trie | (Insert and Search)

Trie is an efficient information re***Trie***val data structure. Using Trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to **M * log N**, where M is maximum string length and N is number of keys in tree. Using Trie, we can search the key in O(M) time. However the penalty is on Trie storage requirements.

Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as end of word node. A Trie node field *isEndOfWord* is used to distinguish the node as end of word node. A simple structure to represent nodes of English alphabet can be as following,

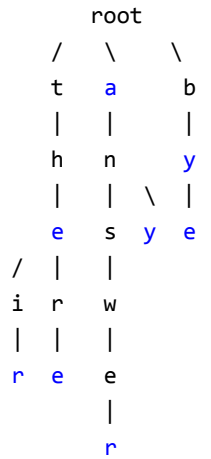
```
// Trie node
struct TrieNode
{
     struct TrieNode *children[ALPHABET_SIZE];

     // isEndOfWord is true if the node
     // represents end of a word
     bool isEndOfWord;
};
```

Inserting a key into Trie is simple approach. Every character of input key is inserted as an individual Trie node. Note that the *children* is an array of pointers (or references) to next level trie nodes. The key character acts as an index into the array *children*. If the input key is new or an extension of existing key, we need to construct non-existing nodes of the key, and mark end of word for last node. If the input key is prefix of existing key in Trie, we simply mark the last node of key as end of word. The key length determines Trie depth.

Searching for a key is similar to insert operation, however we only compare the characters and move down. The search can terminate due to end of string or lack of key in trie. In the former case, if the *isEndofWord* field of last node is true, then the key exists in trie. In the second case, the search terminates without examining all the characters of key, since the key is not present in trie.

The following picture explains construction of trie using keys given in the example below,



In the picture, every character is of type *trie_node_t*. For example, the *root* is of type trie_node_t, and it's children *a*, *b* and *t* are filled, all other nodes of root will be NULL. Similarly, "a" at the next level is having only one child ("n"), all other children are NULL. The leaf nodes are in blue.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Insert and search costs **O(key_length)**, however the memory requirements of Trie is **O(ALPHABET_SIZE * key_length * N)** where N is number of keys in Trie. There are efficient representation of trie nodes (e.g. compressed trie, ternary search tree, etc.) to minimize memory requirements of trie.

C++

```
// C++ implementation of search and insert
// operations on Trie
#include <bits/stdc++.h>
using namespace std;

const int ALPHABET_SIZE = 26;

// trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];

    // isEndOfWord is true if the node represents
    // end of a word
    bool isEndOfWord;
```

```
};

// Returns new trie node (initialized to NULLs)
struct TrieNode *getNode(void)
{
    struct TrieNode *pNode =  new TrieNode;

    pNode->isEndOfWord = false;

    for (int i = 0; i < ALPHABET_SIZE; i++)
        pNode->children[i] = NULL;

    return pNode;
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just
// marks leaf node
void insert(struct TrieNode *root, string key)
{
    struct TrieNode *pCrawl = root;

    for (int i = 0; i < key.length(); i++)
    {
        int index = key[i] - 'a';
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNode();

        pCrawl = pCrawl->children[index];
    }

    // mark last node as leaf
    pCrawl->isEndOfWord = true;
}

// Returns true if key presents in trie, else
// false
bool search(struct TrieNode *root, string key)
{
    struct TrieNode *pCrawl = root;

    for (int i = 0; i < key.length(); i++)
    {
        int index = key[i] - 'a';
        if (!pCrawl->children[index])
            return false;

        pCrawl = pCrawl->children[index];
    }

    return (pCrawl != NULL && pCrawl->isEndOfWord);
}

// Driver
int main()
{
    // Input keys (use only 'a' through 'z'
    // and lower case)
    string keys[] = {"the", "a", "there",
                     "answer", "any", "by",
                      "bye", "their" };
    int n = sizeof(keys)/sizeof(keys[0]);

    struct TrieNode *root = getNode();

    // Construct trie
    for (int i = 0; i < n; i++)
        insert(root, keys[i]);

    // Search for different keys
```

```
    search(root, "the")? cout << "Yes\n" :
                         cout << "No\n";
    search(root, "these")? cout << "Yes\n" :
                           cout << "No\n";
    return 0;
}
```

Run on IDE

C

```
// C implementation of search and insert operations
// on Trie
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

// Alphabet size (# of symbols)
#define ALPHABET_SIZE (26)

// Converts key current character into index
// use only 'a' through 'z' and lower case
#define CHAR_TO_INDEX(c) ((int)c - (int)'a')

// trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];

    // isEndOfWord is true if the node represents
    // end of a word
    bool isEndOfWord;
};

// Returns new trie node (initialized to NULLs)
struct TrieNode *getNode(void)
{
    struct TrieNode *pNode = NULL;

    pNode = (struct TrieNode *)malloc(sizeof(struct TrieNode));

    if (pNode)
    {
        int i;

        pNode->isEndOfWord = false;

        for (i = 0; i < ALPHABET_SIZE; i++)
            pNode->children[i] = NULL;
    }

    return pNode;
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just marks leaf node
void insert(struct TrieNode *root, const char *key)
{
    int level;
    int length = strlen(key);
    int index;

    struct TrieNode *pCrawl = root;
```

```
    for (level = 0; level < length; level++)
    {
        index = CHAR_TO_INDEX(key[level]);
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNode();

        pCrawl = pCrawl->children[index];
    }

    // mark last node as leaf
    pCrawl->isEndOfWord = true;
}

// Returns true if key presents in trie, else false
bool search(struct TrieNode *root, const char *key)
{
    int level;
    int length = strlen(key);
    int index;
    struct TrieNode *pCrawl = root;

    for (level = 0; level < length; level++)
    {
        index = CHAR_TO_INDEX(key[level]);

        if (!pCrawl->children[index])
            return false;

        pCrawl = pCrawl->children[index];
    }

    return (pCrawl != NULL && pCrawl->isEndOfWord);
}

// Driver
int main()
{
    // Input keys (use only 'a' through 'z' and lower case)
    char keys[][8] = {"the", "a", "there", "answer", "any",
                      "by", "bye", "their"};

    char output[][32] = {"Not present in trie", "Present in trie"};


    struct TrieNode *root = getNode();

    // Construct trie
    int i;
    for (i = 0; i < ARRAY_SIZE(keys); i++)
        insert(root, keys[i]);

    // Search for different keys
    printf("%s --- %s\n", "the", output[search(root, "the")] );
    printf("%s --- %s\n", "these", output[search(root, "these")] );
    printf("%s --- %s\n", "their", output[search(root, "their")] );
    printf("%s --- %s\n", "thaw", output[search(root, "thaw")] );

    return 0;
}
```

Run on IDE

Java

```
// Java implementation of search and insert operations
// on Trie
public class Trie {
```

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

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

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

    TrieNode(){
        isEndOfWord = false;
        for (int i = 0; i < ALPHABET_SIZE; i++)
            children[i] = null;
    }
};

static TrieNode root;

// If not present, inserts key into trie
// If the key is prefix of trie node,
// just marks leaf node
static void insert(String key)
{
    int level;
    int length = key.length();
    int index;

    TrieNode pCrawl = root;

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

        pCrawl = pCrawl.children[index];
    }

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

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

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

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

        pCrawl = pCrawl.children[index];
    }

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

// Driver
public static void main(String args[])
{
```

```
        // Input keys (use only 'a' through 'z' and lower case)
        String keys[] = {"the", "a", "there", "answer", "any",
                         "by", "bye", "their"};

        String output[] = {"Not present in trie", "Present in trie"};


        root = new TrieNode();

        // Construct trie
        int i;
        for (i = 0; i < keys.length ; i++)
            insert(keys[i]);

        // Search for different keys
        if(search("the") == true)
            System.out.println("the --- " + output[1]);
        else System.out.println("the --- " + output[0]);

        if(search("these") == true)
            System.out.println("these --- " + output[1]);
        else System.out.println("these --- " + output[0]);

        if(search("their") == true)
            System.out.println("their --- " + output[1]);
        else System.out.println("their --- " + output[0]);

        if(search("thaw") == true)
            System.out.println("thaw --- " + output[1]);
        else System.out.println("thaw --- " + output[0]);

    }
}
// This code is contributed by Sumit Ghosh
```

Run on IDE

Python

```
# Python program for insert and search
# operation in a Trie

class TrieNode:

    # Trie node class
    def __init__(self):
        self.children = [None]*26

        # isEndOfWord is True if node represent the end of the word
        self.isEndOfWord = False

class Trie:

    # Trie data structure class
    def __init__(self):
        self.root = self.getNode()

    def getNode(self):

        # Returns new trie node (initialized to NULLs)
        return TrieNode()

    def _charToIndex(self,ch):

        # private helper function
        # Converts key current character into index
        # use only 'a' through 'z' and lower case
```

```
        return ord(ch)-ord('a')


    def insert(self,key):

        # If not present, inserts key into trie
        # If the key is prefix of trie node,
        # just marks leaf node
        pCrawl = self.root
        length = len(key)
        for level in range(length):
            index = self._charToIndex(key[level])

            # if current character is not present
            if not pCrawl.children[index]:
                pCrawl.children[index] = self.getNode()
            pCrawl = pCrawl.children[index]

        # mark last node as leaf
        pCrawl.isEndOfWord = True

    def search(self, key):

        # Search key in the trie
        # Returns true if key presents
        # in trie, else false
        pCrawl = self.root
        length = len(key)
        for level in range(length):
            index = self._charToIndex(key[level])
            if not pCrawl.children[index]:
                return False
            pCrawl = pCrawl.children[index]

        return pCrawl != None and pCrawl.isEndOfWord

# driver function
def main():

    # Input keys (use only 'a' through 'z' and lower case)
    keys = ["the","a","there","anaswe","any",
            "by","their"]
    output = ["Not present in trie",
              "Present in tire"]

    # Trie object
    t = Trie()

    # Construct trie
    for key in keys:
        t.insert(key)

    # Search for different keys
    print("{} ---- {}".format("the",output[t.search("the")]))
    print("{} ---- {}".format("these",output[t.search("these")]))
    print("{} ---- {}".format("their",output[t.search("their")]))
    print("{} ---- {}".format("thaw",output[t.search("thaw")]))

if __name__ == '__main__':
    main()

# This code is contributed by Atul Kumar (www.facebook.com/atul.kr.007)
```

Run on IDE

Output :

```
the --- Present in trie
these --- Not present in trie
their --- Present in trie
thaw --- Not present in trie
```

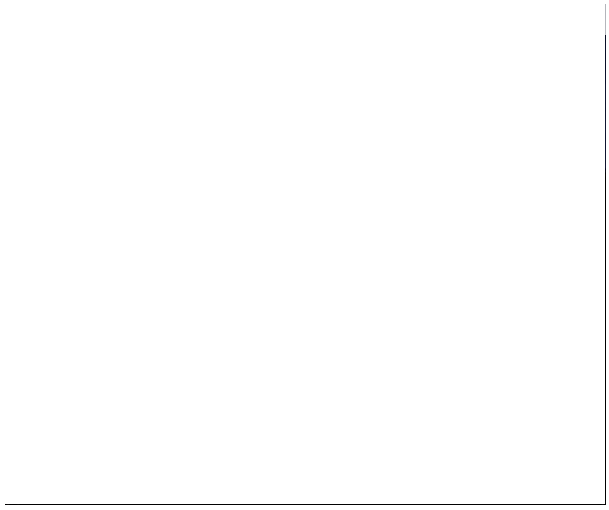
Trie - Insert and Search | GeeksforGeeks

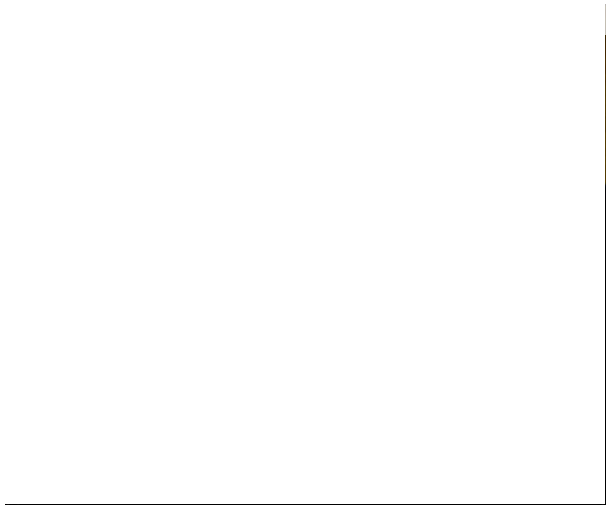
NOTE : In video, **isEndOfWord** is referred as **isLeaf**.

Asked in: DE Shaw

Next Article Trie Delete

This article is contributed by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.





GATE CS Corner Company Wise Coding Practice

Advanced Data Structure TRIE

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

About Venki

Software Engineer

View all posts by Venki →

Recommended Posts:

Trie | (Delete)

Boggle | Set 2 (Using Trie)

Ternary Search Tree
Pattern Searching using a Trie of all Suffixes
Trie memory optimization using hash map
Longest word in ternary search tree
Counting the number of words in a Trie
Order statistic tree using fenwick tree (BIT)
Leftist Tree / Leftist Heap
Trie | (Display Content)

(Login to Rate)

3.5 Average Difficulty : **3.5/5.0**
Based on **103** vote(s)

Basic Easy Medium Hard Expert

Add to TODO List
Mark as DONE

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments Share this post!