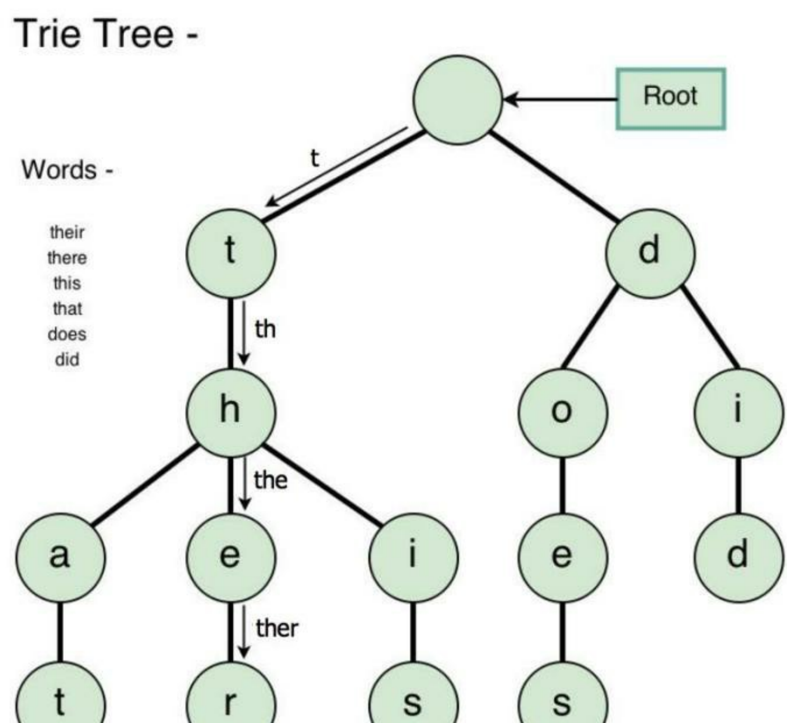
Trie Tree Implementation

JANUARY 16, 2015 / VAMSI SANGAM

Hello people…! In this post we will talk about the Trie Tree Implementation. Trie Trees are are used to search for all occurrences of a word in a given text very quickly. To be precise, if the length of the word is "**L**", the trie tree searches for all occurrences of this data structure in **O(L)** time, which is very very fast in comparison to many pattern matching algorithms.

But I must mention, this data structure is not exactly used for "pattern matching", it is used to search for the occurrences of the word in the given text. How these both functionalities differ…? We'll get to know that shortly. The Trie Tree has many applications. Your browser could be using a trie tree internally to search for words when you press Ctrl + F. So, let's get started with this data structure…!

The Trie Tree is a very straight forward data structure. It is a simple tree where the nodes have an alphabet associated with them. And the way these nodes are arranged is the best part of the Trie Tree. To get an idea take a close look at the sketch below –



RECENT POSTS

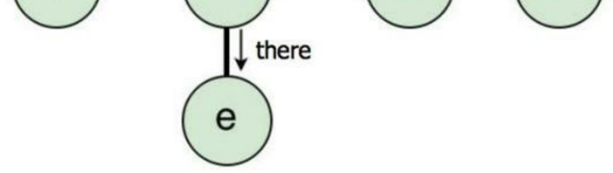
Java Tutorials – Constructor and Overloading Methods March 3, 2016

Encapsulation in Java February 19, 2016

C++ – Variables, Initialization and Assignment January 15, 2016

C++ Programming Style and Structure December 26, 2015

Why should I learn C++? December 23, 2015



Structure of Trie Tree

The arrows in the sketch above indicate how to traverse the trie tree in order to tell if a word exists or not. We travel through the root node, down the tree until we hit a leaf. Picking up a character at every edge, we construct our word. Now, you can easily tell why the time to search for a node in the text will be in the order of length of the word to be searched. It's obvious…! One would have to go down till the leaf. In the worst case, one would have to travel the height of the tree which would be the length of the longest word in the whole text..! 😉

So, as we got a little idea about working with a Trie Tree, let us move on to the serious part. Let us talk about how this is implemented and then we will talk about three fundamental operations done on the Trie Tree –

- Insert
- Delete
- Search

In C, the Trie Tree is implemented using structures. But the C implementation of this data structure gets a little dirty. So we will switch to C++ but we will keep it as C-ish as possible. As we keep discussing about the implementation, you will notice how many advantages we have when we use C++. This will be our structure of each node in the tree –

```
struct node
{
    struct node * parent;
    struct node * children[ALPHABETS];
    vector<int> occurrences;
};
```

- **parent** – This points to the node which is the parent of the current node in the tree hierarchy. It may seem useless at the first, but we need this to travel up the tree when we want to delete any word. We'll get to the deletion operation in a moment.
- **children** – It points to the children nodes. It is made an array so that we can have O(1) accessing time. But why is the size 26…? Its simple. Consider a word, "th", now, what could the third letter possibly be, if it had one…? One among the 26 english alphabets…! So that's why the size is made 26. If you want to make a trie tree for another language, replace the number 26 with the number of letters in your language.
- **occurrences** – This will store the starting indices of the word occurring in the given text. Now, why this is made a vector is that, vector is as good as a Linked List with random access. It is one of the most handy ready made data structures available in the C++ STL Library. If you are not familiar with vectors this is a good place to start.
  If this were C, we would have to give a fixed array size, and we would have a problem if the occurrences of a particular node are more. We could avoid this by putting a Linked List there. But we sacrifice random access and a whole lot of operations get time taking. Moreover, the code will get really really cumbersome to manage if you have a tree and a linked list.

Having got a picture of the implementation, let us look at how the operations are done in a Trie Tree.

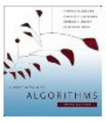
Insert Operation

When we do an insert operation, there are a few cases –

1. The word to be inserted does not exist.
2. The word to be inserted already exists in the tree.

LIKE THIS WEBSITE ON FACEBOOK..!

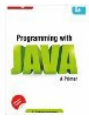
RECOMMENDED READING IN DATA STRUCTURES AND ALGORITHMS

amazon.in

Introduction to Algorithms
INR 771.00
Shop now

amazon.in

Data Structures Using C and...
INR 425.00
Shop now

RECOMMENDED READING FOR JAVA (FOR BEGINNERS)

amazon.in

Programming with Java
INR 300.00
Shop now

amazon.in

Head First Java
INR 700.00
Shop now

RECOMMENDED READING IN JAVA (FOR NON-BEGINNERS)

amazon.in

Java: The Complete...
INR 795.00
Shop now

amazon.in

OCA/OCP Java SE 7...
INR 735.00
Shop now

3. The word to be inserted does not exists, but as the suffix of a word.

The first case is simple. One would have to traverse till the alphabets of the words have nodes in the trie tree or else create new nodes one-after-the-other. And at the end of the word, i.e., the node for the last alphabet, we will mark it as a leaf and push the starting index into the vector indicating the occurrence of the newly inserted word.

During this course of traversal, we will be cutting off the string of the word we have one-by-one as they are processed. This is done by putting using a vector of characters and popping off one character after the other. This is less code to handle and more efficient as we can use a vector as a queue. This is another advantage of using C++.

After having learnt what to do with the first case, you can guess what we would have to do in the second case. We simply have to add a new value to the occurrences vector at the node corresponding to the last alphabet of the word. We can also know the number of occurrences in constant time, we simply return the size of the vector. This is another advantage of using C++.

To understand the challenge in the third case, let's take a simple example. What would you do with your trie tree if you wanted to insert the word "face" if the word "facebook" is already there in your tree…? This is the third case. The answer to this is the occurrence vector itself. We simply push the starting index of the word into the vector of that node which corresponds to the last alphabet of the word to be inserted, in the above example, this would be the node of "e". So, what really tells if there's a word ending with the alphabet corresponding to the current node is the size of the vector.

So I hope you understand how important our vector is. A lot depends on it…!

Delete Operation

The deletion of a word in the trie tree is similar to the insertion, we have a few cases –

- Error 404 : Word not found…!
- Word exists as a stand-alone word.
- Word exists as a prefix of another word.

If the word is not there at all, we don't have to do anything. We just have to make sure that we don't mess up the existing data structure…!

The second case is a little tricky. We would have to delete the word bottom-up. That is, we will delete that part of the word which is not a part of any other word. For example, consider the sketch above. If we were to delete "this", we would delete the letters 'i' and 's' as, 'h' is a part of another word. This keeps us away from distorting the data structure. If the word were existing multiple number of times we will simply remove the occurrence from the vector of the concerned node.

In the third case too, we will simply delete the occurrence of the word from the vector. We needn't write a lot of code as we can use the functions in algorithm header file. This is another advantage of using C++.

Note – When we delete the occurrence of a word, we are not concerned about the validity of the indices stored as occurrences of other words. What I mean to say is, suppose we have 10 words. If we delete the 3rd word, the 5th word or the 9th word is supposed to become the 4rth and the 8th word as far as the original text is concerned. But we will not consider this. The data stored in the trie tree is not meant to me deleted or inserted. The Trie Tree is meant for processing the given text not to manipulate the given text.

Search Operation

The search operation is simple and is as we discussed when we began our discussion about the Trie Tree. We go down the tree traversing the nodes and keep "picking up characters" as we go. And the occurrences vector tells us if a word exists that ends with the alphabet associated with the current node, and if so, it gives us the indices of occurrences and also the number of occurrences.

Besides these basic operations, there is another very interesting operation that is done with the Trie Tree –

- **Lexicographical Sort** – If we want to print all the words processed into the trie tree lexicographically, all we have to do is do a Preorder Walk on the tree. This will automatically print all the words in the lexicographical order or the dictionary order. This is due to the very structure and arrangement of nodes in a Trie Tree. Now, I'd like to share another interesting thing about pre-order walk in trees… The Pre-order walk works exactly as a Depth First Search (DFS) in graphs, i.e., the sequence in which both the algorithms visit the nodes is the same. Think about this for a while and word out the two algorithms on an example (you could take mine in the sketch above), and you can see why it is so. You will also notice why the printed words would be lexicographically sorted.

Now, having learned a lot about the trie tree, try coding it in C++. If you are uneasy with C++, you can try it in C, but make sure you try at least 3 times. Trying is very important. I don't know if you are new to reading my posts, but I insist a lot on trying in every post of mine…! If you have succeeded, you're fabulous…! If not, check out my code below any try figuring out how just close you were…!!

```cpp
/* ==========  ========== ========== ========= */
//          Trie Tree Data Structure           //
//                using C++ STL                //
//                                             //
//         Functions follow Pascal Case        //
//           Convention and Variables          //
//         follow Camel Case Convention        //
//                                             //
//            Author - Vamsi Sangam            //
//            Theory of Programming            //
/* ========== ========== ========== ========== */

#include <cstdio>
#include <cstdlib>
#include <vector>

#define ALPHABETS 26
#define CASE 'a'
#define MAX_WORD_SIZE 25

using namespace std;

struct Node
{
    struct Node * parent;
    struct Node * children[ALPHABETS];
    vector<int> occurrences;
};

// Inserts a word 'text' into the Trie Tree
// 'trieTree' and marks it's occurence as 'index'.
void InsertWord(struct Node * trieTree, char * word, int ind
{
    struct Node * traverse = trieTree;

    while (*word != '\0') {     // Until there is something
        if (traverse->children[*word - CASE] == NULL) {
            // There is no node in 'trieTree' corresponding

            // Allocate using calloc(), so that components a
            traverse->children[*word - CASE] = (struct Node
            traverse->children[*word - CASE]->parent = trave
        }

        traverse = traverse->children[*word - CASE];
        ++word; // The next alphabet
    }

    traverse->occurrences.push_back(index);      // Mark the
}

// Searches for the occurence of a word in 'trieTree',
// if not found, returns NULL,
// if found, returns poniter pointing to the
// last node of the word in the 'trieTree'
// Complexity -> O(length_of_word_to_be_searched)
struct Node * SearchWord(struct Node * treeNode, char * word
{
    // Function is very similar to insert() function
    while (*word != '\0') {
        if (treeNode->children[*word - CASE] != NULL) {
            treeNode = treeNode->children[*word - CASE];
            ++word;
        } else {
            break;
        }
    }

    if (*word == '\0' && treeNode->occurrences.size() != 0)
        // Word found
        return treeNode;
    } else {
        // Word not found
        return NULL;
    }
}

// Searches the word first, if not found, does nothing
// if found, deletes the nodes corresponding to the word
void RemoveWord(struct Node * trieTree, char * word)
{
    struct Node * trieNode = SearchWord(trieTree, word);

    if (trieNode == NULL) {
        // Word not found
        return;
    }

    trieNode->occurrences.pop_back();    // Deleting the occ

    // 'noChild' indicates if the node is a leaf node
    bool noChild = true;

    int childCount = 0;
    // 'childCount' has the number of children the current n
    // has which actually tells us if the node is associated
    // another word .This will happen if 'childCount' != 0.
    int i;

    // Checking children of current node
    for (i = 0; i < ALPHABETS; ++i) {
        if (trieNode->children[i] != NULL) {
            if (trieNode.children[i] != null) {
```

```
            noChild = false;
            ++childCount;
        }
    }

    if (!noChild) {
        // The node has children, which means that the word
        // occurrence was just removed is a Suffix-Word
        // So, logically no more nodes have to be deleted
        return;
    }

    struct Node * parentNode;     // variable to assist in t

    while (trieNode->occurrences.size() == 0 && trieNode->pa
        // trieNode->occurrences.size() -> tells if the node
        //
        // trieNode->parent != NULL -> is the base case sort
        // out of nodes to be deleted, as we reached the roo
        //
        // childCount -> does the same thing as explained in
        // we reach

        childCount = 0;
        parentNode = trieNode->parent;

        for (i = 0; i < ALPHABETS; ++i) {
            if (parentNode->children[i] != NULL) {
                if (trieNode == parentNode->children[i]) {
                    // the child node from which we reached
                    // the parent, this is to be deleted
                    parentNode->children[i] = NULL;
                    free(trieNode);
                    trieNode = parentNode;
                } else {
                    ++childCount;
                }
            }
        }
    }
}

// Prints the 'trieTree' in a Pre-Order or a DFS manner
// which automatically results in a Lexicographical Order
void LexicographicalPrint(struct Node * trieTree, vector<cha
{
    int i;
    bool noChild = true;

    if (trieTree->occurrences.size() != 0) {
        // Condition trie_tree->occurrences.size() != 0,
        // is a neccessary and sufficient condition to
        // tell if a node is associated with a word or not

        vector<char>::iterator charItr = word.begin();

        while (charItr != word.end()) {
            printf("%c", *charItr);
            ++charItr;
        }
        printf(" -> @ index -> ");

        vector<int>::iterator counter = trieTree->occurrence
        // This is to print the occurences of the word

        while (counter != trieTree->occurrences.end()) {
            printf("%d, ", *counter);
            ++counter;
        }

        printf("\n");
    }

    for (i = 0; i < ALPHABETS; ++i) {
        if (trieTree->children[i] != NULL) {
            noChild = false;
            word.push_back(CASE + i);   // Select a child

            // and explore everything associated with the ci
            LexicographicalPrint(trieTree->children[i], word
            word.pop_back();
            // Remove the alphabet as we dealt
            // everything associated with it
        }
    }

    word.pop_back();
}

int main()
{
    int n, i;
    vector<char> printUtil;       // Utility variable to pri

    // Creating the Trie Tree using calloc
    // so that the components are initialised
    struct Node * trieTree = (struct Node *) calloc(1, sizeo
    char word[MAX_WORD_SIZE];

    printf("Enter the number of words-\n");
    scanf("%d", &n);
    for (i = 1; i <= n; ++i) {
        scanf("%s", word);
        InsertWord(trieTree, word, i);
    }

    printf("\n");   // Just to make the output more readable
    LexicographicalPrint(trieTree, printUtil);

    printf("\nEnter the Word to be removed - ");
    scanf("%s", word);
    RemoveWord(trieTree, word);

    printf("\n");   // Just to make the output more readable
    LexicographicalPrint(trieTree, printUtil);

    return 0;
}
```

The code is highly commented with explanation. It is well described, but if you have any doubts regarding the data structure or the code, feel free to comment them. I have used a few macros. The macro *CASE* indicates for which case the Trie Tree works. If we mention 'A' as the macro, the Trie Tree will work for upper case words only.

Other Implementations

- Trie Tree using C++ Classes

The code is well tested against fairly large input. You can download the test case file here – Trie Tree Input **(PDF)**. You can just clock your code for the insert operations. My code took 1.236 seconds to execute that loop which reads a word and inserts it into the Trie Tree. There are 5000 words in total. The last word in the input file is the word to be deleted.

If you think you can give a suggestion to make my code better, please do comment them too. I appreciate your suggestions. For those there who are struggling to get their code right, "Keep the effort going guys"…! Remember, you won't learn anything if you keep writing Hello World program again and again. So, keep practicing…! I hope my post has helped you in getting to know about the Trie Tree. If it did, let me know by commenting! Happy Coding…! 😀

Related

Trie Tree Practise - SPOJ - PHONELST	Trie Tree Practise - SPOJ - DICT	Segment Trees
August 24, 2015	September 1, 2015	January 27, 2015
In "SPOJ"	In "Competitive Coding"	In "Tree Data Structures"

Tree Data Structures

DATA STRUCTURES · PATTERN MATCHING · STRINGS · TRIE TREES

← Dijkstra's Algorithm

Bellman Ford Algorithm →

24 thoughts on "Trie Tree Implementation"

Lakshay Virmani

JANUARY 20, 2016 AT 6:35 PM

Thanks a lot man! Gr8 explanation. Much better than gfg. Loved it. 😀

REPLY

Vamsi Sangam

JANUARY 21, 2016 AT 1:30 PM

Thanks a lot..! I'm glad you liked it 🙂

REPLY

Chhavi P. Gupta

JANUARY 5, 2016 AT 2:42 AM

Hey, I loved your code really, just one doubt, in DeleteWord function, shouldn't childcount be reinitialised to 0 inside the last while loop for every node we go up to? And shouldn't it be incremented only if the child is not equal to the node we are abt to delete? Because otherwise even after deleting the child we will have childcount as 1 which will make us exit the loop even tho there were no other children nodes of that parent node.

REPLY

Vamsi Sangam

JANUARY 10, 2016 AT 10:01 PM

Yup, you are right… Silly mistake… Thanks for pointing it out…! Corrected the code 🙂

REPLY

LJ

NOVEMBER 9, 2015 AT 11:53 PM

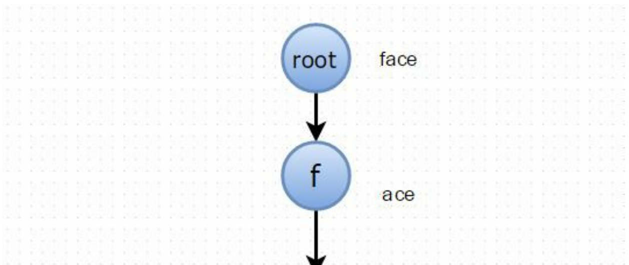
Hey,
Really nice job. But could you explain the place where you are using CASE in adding word? 🙂

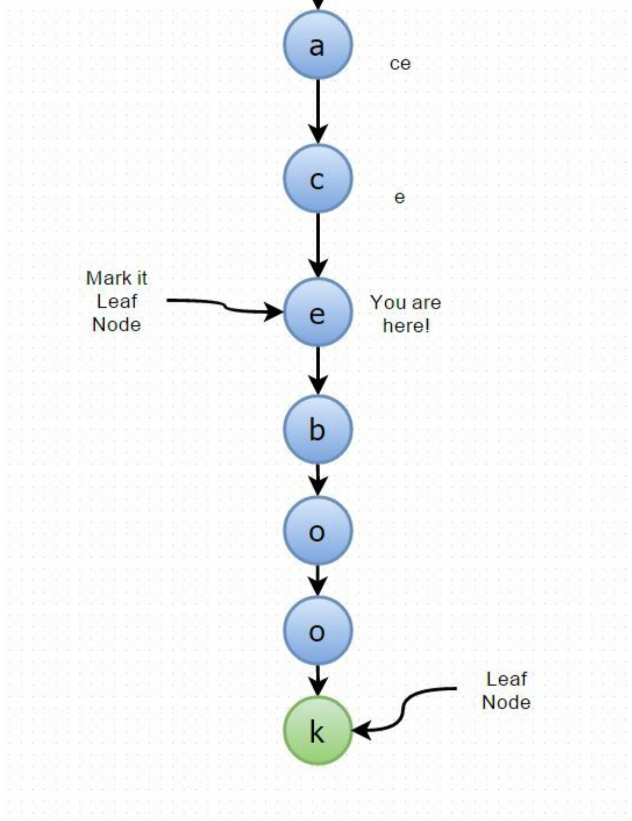
REPLY

Vamsi Sangam

NOVEMBER 13, 2015 AT 9:06 PM

Well, I think you meant "face"… If you really meant "case", then there's nothing great, it comes under the category, "word to be inserted does not exist"…. When we are inserting "face" (when we already inserted "facebook").. Then, when you reach the node "e" while traversing to insert "face", you will see that "e" is a node which has a child but is not a leaf node… Well, let's forget about the vectors in our struct, let's say that there's simply a boolean variable isLeaf, in your struct which tells you if it is a word ending… Look at the diagram below –





You see that I have marked "k" as a leaf node… Not exactly because it is a leaf for that tree, but because we have a word "facebook", now as we have to insert a word "face", we will mark "e" as a leaf node. Now if you want to do such operations in our struct which has a vector, we simply push its occurrence to the vector…. So the size of the vector indirectly tells us if the node is a leaf node or not…. Think about it for a while… Give it sometime… I'm sure you'll get it 😀

REPLY

Mayank Pratap

OCTOBER 20, 2015 AT 8:51 AM

I am yet a beginner in trie but I think in the example tree ordering of childs is a bit wrong…For eg:-ordering of a,e,i is different from ordering of d and t.

REPLY

Vamsi Sangam

OCTOBER 20, 2015 AT 4:18 PM

Yeah, you are right…. It should be A to Z… I overlooked it… I will correct it as soon as possible.. 🙂 … Thanks for pointing it out 😀

REPLY

Saurabh

JUNE 27, 2015 AT 2:11 PM

There's one doubt I am having though.
How is a trie useful if one has to atleast read all the characters from the input array of strings to insert the strings one by one into the trie?So the complexity is O(size_of_array).Suppose the input array of strings is {"hello",world","stack","overflow"}.And we want to search for "stack",then we would have to atleast traverse the whole array for inserting the keys into the

trie.So complexity is O(size_of_array).We could do this without a trie.
Please help.
Thanks.

REPLY

Vamsi Sangam

JUNE 28, 2015 AT 10:36 PM

Ok… I think I get your argument… For the sake of simplicity, let's say we have 'N' words of length 'L'… In the case of a 2D array… Getting the data structure ready will cost us O(N) time… And subsequent searches for any word will cost us O(NL) time… Now, getting a trie tree ready will cost us O(NL) time, but the subsequent searches cost us only O(L) time… And that is what we are interested in… The fast search… If your requirement demands that you must perform too many searches for words… Then you must think of a trie tree… You don't construct a trie tree if you want to search only once…. So it is all in the number of operations you want to perform… Let's say, if you wanted to perform a 100 or more searches for a word.. Then go for a trie tree..! Let me know if you have any more doubts.. 🙂

REPLY

Saurabh

JUNE 26, 2015 AT 10:42 PM

Thanks dude,you made my day…

REPLY

Vamsi Sangam

JUNE 26, 2015 AT 11:23 PM

My pleasure…! 😀

REPLY

sanjeev

JUNE 24, 2015 AT 7:43 PM

thanks man nice and clean explanation 🙂

REPLY

Vamsi Sangam

JUNE 25, 2015 AT 12:11 AM

Thanks a lot..! 😀

REPLY

Aashay

JUNE 22, 2015 AT 12:36 PM

In lexicographical method , temp which is a empty vector sending to the

function and
itr is pointing to the begining to that vector word and it printing the character till its end
but my question is its empty how it will print the character for suppose if root is a word whith occurence.size != 0; then how that line 158 -163 will print anything ????

REPLY

Vamsi Sangam

JUNE 22, 2015 AT 7:18 PM

Nice q'..! Well… If the root is a word with occurences.size != 0, then it would mean that the word is an empty word → ""… Any non-empty word would definitely lead to an edge. I modified a small flaw in my trie tree diagram to support my argument… You see, an edge represents an alphabet in the word… If there is an empty word… then word.begin() would be equal to word.end(), and it would not print anything as it should not because the word is ""… And the occurrence is printed… Such an output is perfectly valid… I hope this clears your doubts… Let me know if you have anymore issues.. 🙂

REPLY

competitivecoder

JUNE 22, 2015 AT 1:01 AM

Wowwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww………
🙂 🙂 I am in love with your blog

REPLY

Vamsi Sangam

JUNE 22, 2015 AT 10:58 AM

Haha..! 😀 … Thanks a lot..!! 😀

REPLY

Juan Carlos Aravena Esparza

JUNE 7, 2015 AT 6:56 AM

There's an error in the lexicographicPrint function.

If you have strings like: "a", "b", "ab", "ba" it wil print "ab", "a", "ba", "b", but the correct output should be "a", "ab", "b", "ba".

To fix it you just have to place the recursive call after the prints.

Saludos and thanks for this code.

REPLY

Vamsi Sangam

JUNE 7, 2015 AT 10:37 AM

Oh yes… I guess is was a bit careless about the lexicographical order…

😛 … Thanks a lot for pointing out the error Carlos..! 🙂

REPLY

Reza

JUNE 5, 2015 AT 12:04 PM

Hi
Thank For This Post
I want to translate this c code to c++
i mean i want to change struct with class and …
can you help me ?

REPLY

Vamsi Sangam

JUNE 6, 2015 AT 8:17 AM

Hi Reza..! I'd really like to help you… But, I don't have much experience in OOP with C++… However, I tried to code it using a class which has a few variables and methods related to Trie Tree… Check this page… Trie Tree using C++ Class.

I guess you will still find it like a C program… 😛 … Do correct my style of coding by commenting them..! 🙂

REPLY

Rupesh Maity

FEBRUARY 7, 2015 AT 5:27 AM

Thank you for this wonderful explanation. Wanted to learn this since a long time.

REPLY

Vamsi Sangam

FEBRUARY 7, 2015 AT 5:32 AM

I am happy that my post helped you, thanks for letting me know Rupesh….! ☺️

REPLY

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

Post Comment

☐ Notify me of follow-up comments by email.

☐ Notify me of new posts by email.

CATEGORIES

Select Category

ARCHIVES

Select Month

SUBSCRIBE TO BLOG VIA EMAIL

Enter your email address to subscribe to this blog and receive notifications of new posts by email.
Join 120 other subscribers

Email Address

Subscribe

PROUDLY POWERED BY WORDPRESS | THEME: HEMINGWAY REWRITTEN BY ANDERS NORÉN. | PRIVACY POLICY