GeeksforGeeks
A computer science portal for geeks

Custom Search

Practice GATE CS Placements Videos Contribute

Login/Register

Trie | (Delete)

In the previous post on trie we have described how to insert and search a node in trie. Here is an algorithm how to delete a node from trie.

During delete operation we delete the key in bottom up manner using recursion. The following are possible conditions when deleting key from trie,

1. Key may not be there in trie. Delete operation should not modify trie.
2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in trie). Delete all the nodes.
3. Key is prefix key of another long key in trie. Unmark the leaf node.
4. Key present in trie, having atleast one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

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

The highlighted code presents algorithm to implement above conditions. (One may be in dilemma how a pointer passed to delete helper is reflecting changes from deleteHelper to deleteKey. Note that we are holding trie as an ADT in trie_t node, which is passed by reference or pointer).

C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

// Alphabet size (# of symbols)

#define ALPHABET_SIZE (26)
#define INDEX(c) ((int)c - (int)'a')
```

```
#define FREE(p) \
    free(p);    \
    p = NULL;

// forward declration
typedef struct trie_node trie_node_t;

// trie node
struct trie_node
{
    int value; // non zero if leaf
    trie_node_t *children[ALPHABET_SIZE];
};

// trie ADT
typedef struct trie trie_t;

struct trie
{
    trie_node_t *root;
    int count;
};

trie_node_t *getNode(void)
{
    trie_node_t *pNode = NULL;

    pNode = (trie_node_t *)malloc(sizeof(trie_node_t));

    if( pNode )
    {
        int i;

        pNode->value   = 0;

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

    return pNode;
}

void initialize(trie_t *pTrie)
{
    pTrie->root = getNode();
    pTrie->count = 0;
}

void insert(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pTrie->count++;
    pCrawl = pTrie->root;

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

        if( pCrawl->children[index] )
        {
            // Skip current node
            pCrawl = pCrawl->children[index];
        }
        else
        {
            // Add new node
            pCrawl->children[index] = getNode();
            pCrawl = pCrawl->children[index];
        }
    }

    // mark last node as leaf (non zero)
    pCrawl->value = pTrie->count;
}

int search(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pCrawl = pTrie->root;

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

        if( !pCrawl->children[index] )
        {
            return 0;
        }

        pCrawl = pCrawl->children[index];
    }

    return (0 != pCrawl && pCrawl->value);
}

int leafNode(trie_node_t *pNode)
{
    return (pNode->value != 0);
}

int isItFreeNode(trie_node_t *pNode)
{
    int i;
    for(i = 0; i < ALPHABET_SIZE; i++)
    {
        if( pNode->children[i] )
            return 0;
    }

    return 1;
}

bool deleteHelper(trie_node_t *pNode, char key[], int level, int len)
{
    if( pNode )
    {
        // Base case
        if( level == len )
        {
            if( pNode->value )
            {
                // Unmark leaf node
                pNode->value = 0;

                // If empty, node to be deleted
                if( isItFreeNode(pNode) )
                {
                    return true;
                }
                return false;
            }
        }
        else // Recursive case
        {
            int index = INDEX(key[level]);

            if( deleteHelper(pNode->children[index], key, level+1, len) )
            {
                // last node marked, delete it
                FREE(pNode->children[index]);

                // recursively climb up, and delete eligible nodes
                return ( !leafNode(pNode) && isItFreeNode(pNode) );
            }
        }
    }

    return false;
}

void deleteKey(trie_t *pTrie, char key[])
{
    int len = strlen(key);

    if( len > 0 )
    {
        deleteHelper(pTrie->root, key, 0, len);
    }
}

int main()
{
    char keys[][8] = {"she", "sells", "sea", "shore", "the", "by", "sheer"};
    trie_t trie;

    initialize(&trie);

    for(int i = 0; i < ARRAY_SIZE(keys); i++)
    {
        insert(&trie, keys[i]);
    }

    deleteKey(&trie, keys[0]);

    printf("%s %s\n", "she", search(&trie, "she") ? "Present in trie" : "Not present in trie"

    return 0;
}
```

Run on IDE

Python

```
# Python program for delete operation
# in a Trie

class TrieNode(object):
    '''
    Trie node class
    '''
    def __init__(self):
        self.children  = [None]*26
```

```python
        # non zero if leaf
        self.value = 0

    def leafNode(self):
        '''
        Check if node is leaf node or not
        '''
        return self.value != 0

    def isItFreeNode(self):
        '''
        If node have no children then it is free
        If node have children return False else True
        '''
        for c in self.children:
            if c:return False
        return True


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

        # keep count on number of keys
        # inserted in trie
        self.count = 0;

    def _Index(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 getNode(self):
        '''
        Returns new trie node (initialized to NULLs)
        '''
        return TrieNode()

    def insert(self,key):
        '''
        If not present, inserts key into trie
        If the key is prefix of trie node,mark
        it as leaf(non zero)
        '''
        length = len(key)
        pCrawl = self.root
        self.count += 1

        for level in range(length):
            index = self._Index(key[level])

            if pCrawl.children[index]:
                # skip current node
                pCrawl = pCrawl.children[index]
            else:
                # add new node
                pCrawl.children[index] = self.getNode()
                pCrawl = pCrawl.children[index]

        # mark last node as leaf (non zero)
        pCrawl.value = self.count

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

        return pCrawl != None and pCrawl.value != 0


    def _deleteHelper(self,pNode,key,level,length):
        '''
        Helper function for deleting key from trie
        '''
        if pNode:
            # Base case
            if level == length:
                if pNode.value:
                    # unmark leaf node
                    pNode.value = 0

                # if empty, node to be deleted
                return pNode.isItFreeNode()

            # recursive case
            else:
                index = self._Index(key[level])
                if self._deleteHelper(pNode.children[index],\
                                        key,level+1,length):

                    # last node marked,delete it
                    del pNode.children[index]

                    # recursively climb up and delete
                    # eligible nodes
                    return (not pNode.leafNode() and \
                                pNode.isItFreeNode())

        return False

    def deleteKey(self,key):
        '''
        Delete key from trie
        '''
        length = len(key)
        if length > 0:
            self._deleteHelper(self.root,key,0,length)


def main():
    keys = ["she","sells","sea","shore","the","by","sheer"]
    trie = Trie()
    for key in keys:
        trie.insert(key)

    trie.deleteKey(keys[0])

    print("{} {}".format(keys[0],\
        "Present in trie" if trie.search(keys[0]) \
                        else "Not present in trie"))

    print("{} {}".format(keys[6],\
        "Present in trie" if trie.search(keys[6]) \
                        else "Not present in trie"))

if __name__ == '__main__':
    main()

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

Run on IDE

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

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 | (Insert and Search)
AVL Tree | Set 1 (Insertion)
How to Implement Reverse DNS Look Up Cache?
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)

3.8 Average Difficulty : **3.8/5.0**
Based on **44** 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!

Contact Us! About Us! Careers! Privacy Policy