

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.

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).

```
#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)
{
    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,
```

```

    {
        // 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", "till"};
    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" : "Not Present");

    return 0;
}
```