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 leaf node. A trie node field *value* will be used to distinguish the node as leaf node (there are other uses of the *value* field). A simple structure to represent nodes of English alphabet can be as following,

```
struct trie_node
{
    int value; /* Used to mark leaf nodes */
    trie_node_t *children[ALPHABET_SIZE];
};
```

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 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 leaf node. If the input key is prefix of existing key in trie, we simply mark the last node of key as leaf. 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 *value* field of last node is non-zero 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,

```
                       root
                    /   \    \
                    t   a     b
                    |   |     |
                    h   n     y
                    |   |  \  |
                    e   s  y  e
                 /  |   |
                 i  r   w
                 |  |   |
                 r  e   e
                        |
                        r
```

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.

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.

```
#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)
// 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
typedef struct trie_node trie_node_t;
struct trie_node
{
    int value;
    trie_node_t *children[ALPHABET_SIZE];
};

// trie ADT
typedef struct trie trie_t;
struct trie
{
    trie_node_t *root;
    int count;
};

// Returns new trie node (initialized to NULLs)
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;
}

// Initializes trie (root is dummy node)
void initialize(trie_t *pTrie)
{
    pTrie->root = getNode();
    pTrie->count = 0;
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just marks leaf no
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 = CHAR_TO_INDEX(key[level]);
        if( !pCrawl->children[index] )
        {
            pCrawl->children[index] = getNode();
        }

        pCrawl = pCrawl->children[index];
    }

    // mark last node as leaf
    pCrawl->value = pTrie->count;
}

// Returns non zero, if key presents in trie
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 = CHAR_TO_INDEX(key[level]);

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

        pCrawl = pCrawl->children[index];
    }

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

// Driver
int main()
{
    // Input keys (use only 'a' through 'z' and lower ca
    char keys[][8] = {"the", "a", "there", "answer", "an
    trie_t trie;

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

    initialize(&trie);

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

    // Search for different keys
    printf("%s --- %s\n", "the", output[search(&trie, "t
    printf("%s --- %s\n", "these", output[search(&trie,
    printf("%s --- %s\n", "their", output[search(&trie,
    printf("%s --- %s\n", "thaw", output[search(&trie, "

    return 0;
}
```