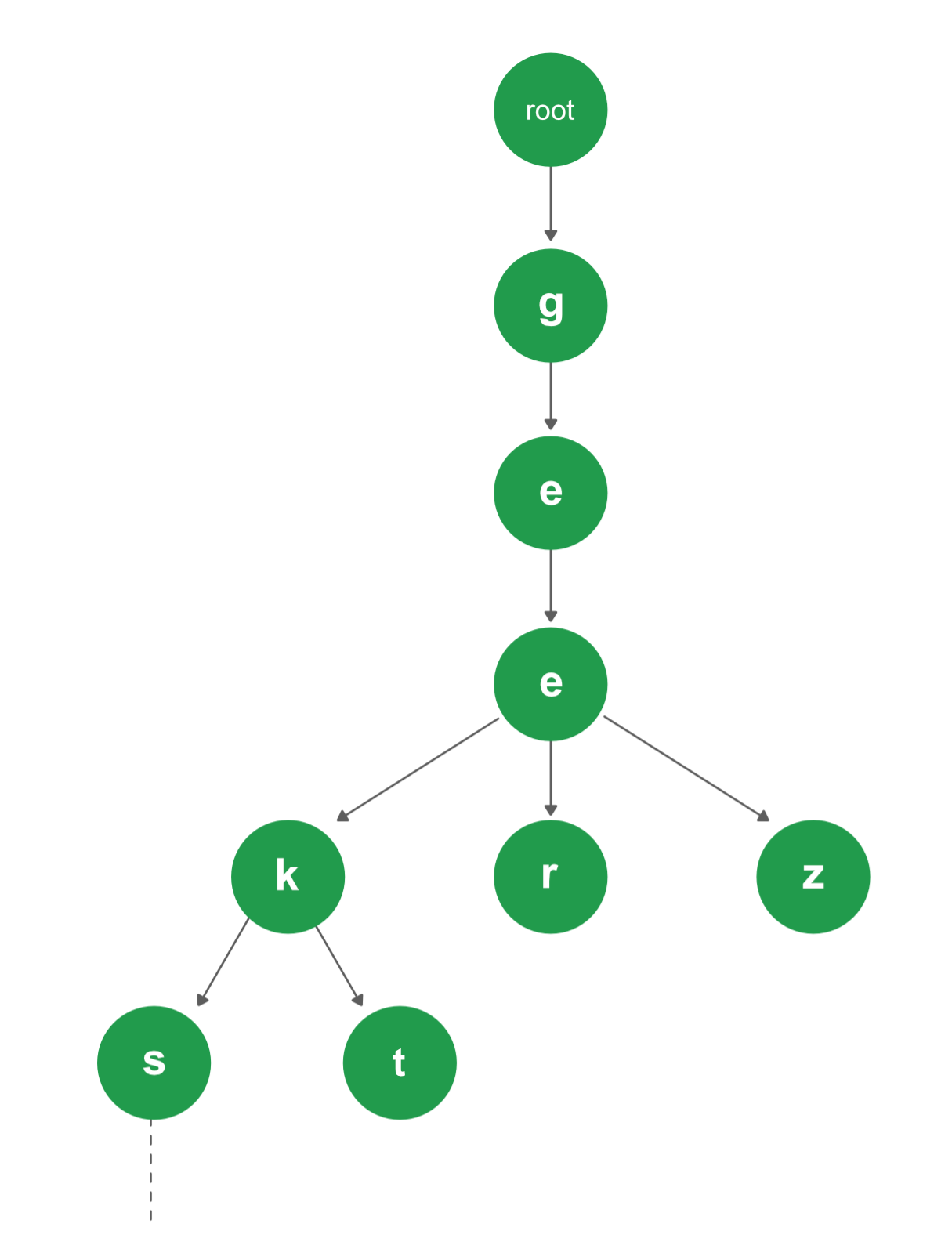
Trie Data Structure

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 keys 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 the 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;

};

**Insert into Trie**

Inserting a key into Trie is a simple approach. Every character of the 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 the existing key, we need to construct non-existing nodes of the key, and mark end of the word for the last node. If the input key is a prefix of the existing key in Trie, we simply mark the last node of the key as the end of a 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 the end of a string or lack of key in the Trie. In the former case, if the isEndofWord field of the last node is true, then the key exists in the Trie.

In the second case, the search terminates without examining all the characters of the key, since the key is not present in the 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. Similary, “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.

// 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;

}

**Output:**

the --- Present in trie

these --- Not present in trie

their --- Present in trie

thaw --- Not present in trie

Delete 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 at least one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

The below code represents algorithm to implement above conditions.

// C implementation of delete operations on Trie

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#define 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 = NULL;

pNode = (struct TrieNode \*)malloc(sizeof(struct 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, char key[]) {

struct TrieNode\* pCrawl = root;

for (int i = 0; key[i]; 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, char key[]) {

struct TrieNode\* pCrawl = root;

for (int i = 0; key[i]; i++) {

int index = key[i] - 'a';

if (!pCrawl->children[index])

return false;

pCrawl = pCrawl->children[index];

}

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

}

// Returns true if root has no children, else false

bool isEmpty(struct TrieNode\* root) {

for (int i = 0; i < ALPHABET\_SIZE; i++)

if (root->children[i])

return false;

return true;

}

// Recursive function to delete a key from given Trie

struct TrieNode\* deleteNode(struct TrieNode\* root, char key[], int depth) {

// If tree is empty

if (!root)

return NULL;

// If last character of key is being processed

if (depth == strlen(key)) {

// This node is no more end of word after

// removal of given key

if (root->isEndOfWord)

root->isEndOfWord = false;

// If given is not prefix of any other word

if (isEmpty(root)) {

free(root);

root = NULL;

}

return root;

}

// If not last character, recur for the child

// obtained using ASCII value

int index = key[depth] - 'a';

root->children[index] =

deleteNode(root->children[index], key, depth + 1);

// If root does not have any child (its only child got

// deleted), and it is not end of another word.

if (isEmpty(root) && root->isEndOfWord == false) {

free(root);

root = NULL;

}

return root;

}

// Driver

int main() {

// Input keys (use only 'a' through 'z'

// and lower case)

char keys[][10] = { "the", "a", "there",

"answer", "any", "by",

"bye", "their", "hero", "heroplane" };

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") ? printf("Yes\n") : printf("No\n");

search(root, "these") ? printf("Yes\n") : printf("No\n");

deleteNode(root, "heroplane", 0);

search(root, "hero") ? printf("Yes\n") : printf("No\n");

return 0;

}

**Output:**

Yes

No

Yes