Trie Data Structure

---

https://www.codechef.com/START70B/problems/OPERATION- First Problem

Intro

- Trie is a type of k-ary search tree used for storing and searching a specific key from a set.
- Using Trie, search complexities can be brought to optimal limit (key length).
- If we store keys in a binary search tree, a well balanced BST will need time proportional to $M * logN$, where **M** is the maximum string length and N is the number of keys in the tree.
- Using Trie, the key can be searched in $O(M)$ time. However, the penalty is on Trie storage requirements

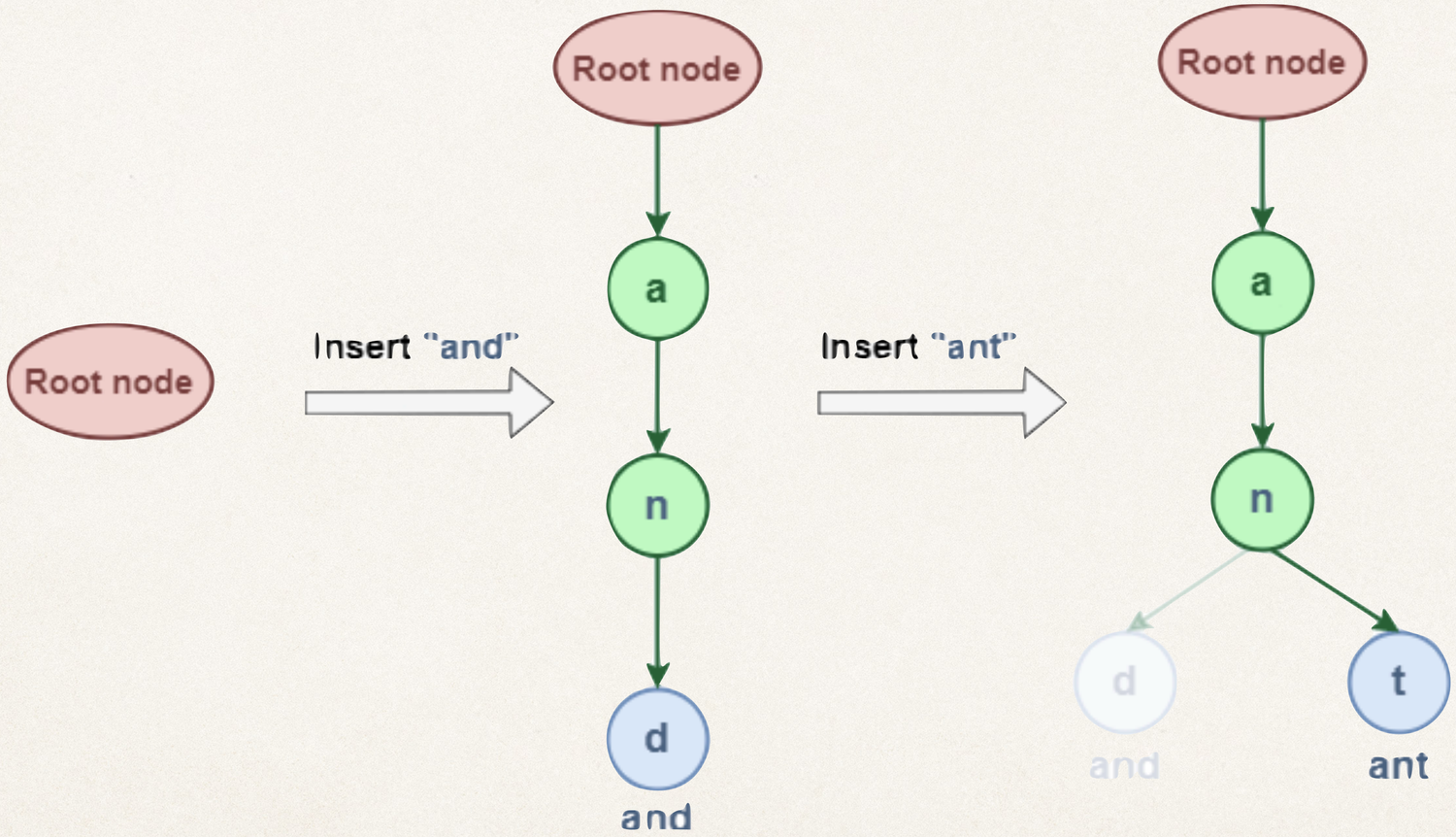
Inserting key into trie

- 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 to the array children.
- If the input key is new or an extension of the existing key, construct non-existing nodes of the key, and mark the end of the word for the last node.

- If the input key is a prefix of the existing key in Trie, Simply mark the last node of the key as the end of a word.

- The key length determines Trie depth.
- The following picture explains the construction of trie using keys given in the example below.



Searching

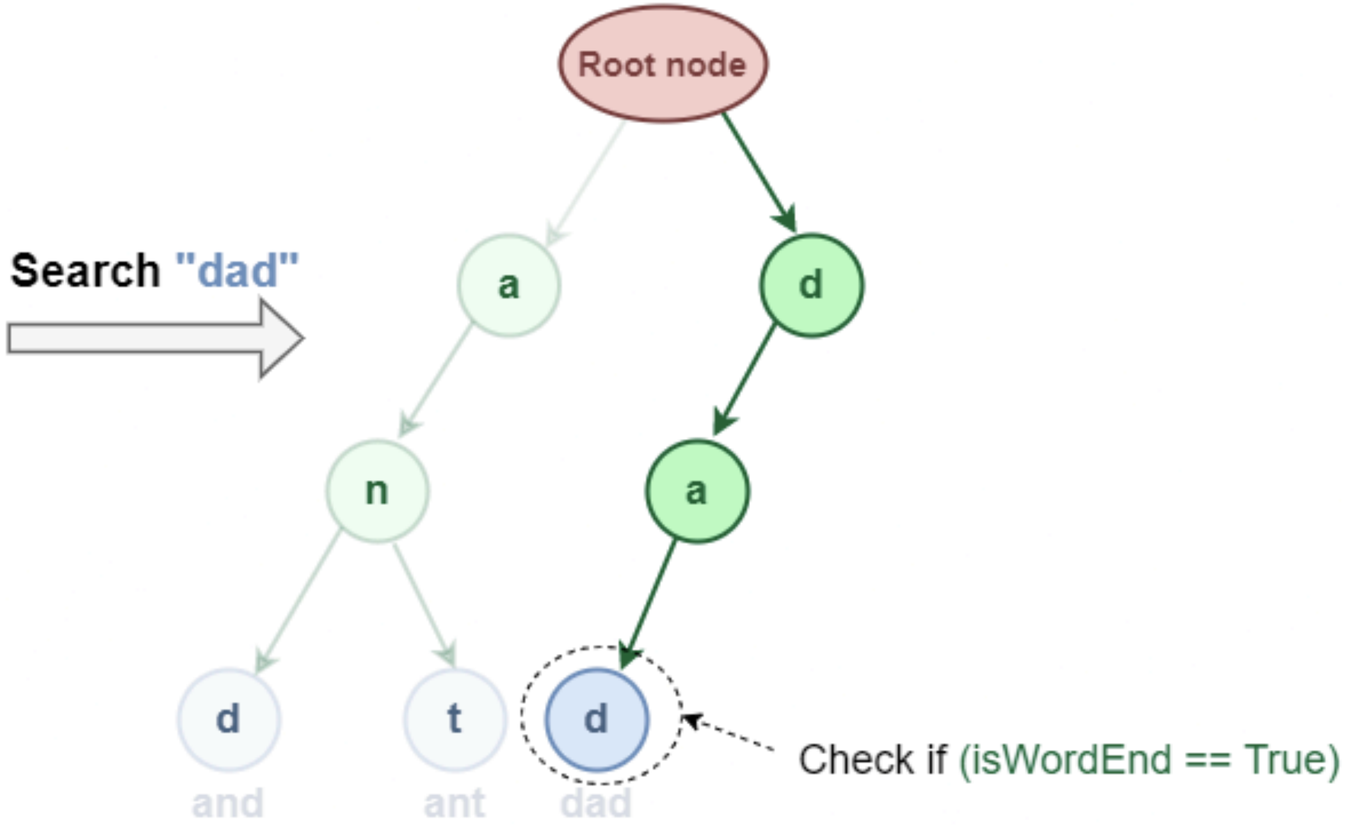
Search Operation in Trie:

Searching for a key is similar to the insert operation.

However, It only **compares the characters and moves 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.



```
struct Node{

    Node *links[26];
    bool flag =false;
    bool containsKey(char ch){
        return (links[ch-'a']≠NULL);
    }
    void put(char ch,Node *node){
        links[ch-'a']=node;
    }
    void setEnd(){
        flag=true;
    }
    bool isEnd(){
        return flag;
    }
    Node* get(char ch){
        return links[ch-'a'];
    }
};
```

```
class Trie {
private:
    Node* root;
public:

    /** Initialize your data structure here. */
    Trie() {
        root = new Node();
    }

    /** Inserts a word into the trie. */

    void insert(string word) {
        Node *node=root;
        for(int i=0;i<word.size();i++){
            if(!node->containsKey(word[i])){
                node->put(word[i],new Node());
            }
            //move to the reference trie
            node=node->get(word[i]);
        }
        node->setEnd();
    }
```

```
    /** Returns if the word is in the trie. */
    bool search(string word) {
        Node *node= root;
        for(int i=0;i<word.size();i++){
            if(!node->containsKey(word[i])){
                return false;
            }
            node= node->get(word[i]);
        }
        return node->isEnd();
    }

    /** Returns if there is any word in the trie that starts with the given prefix. */
    bool startsWith(string word) {
        Node *node =root;
        for(int i=0;i<word.length();i++){
            if(!node->containsKey(word[i])){
                return false;
            }
            node=node->get(word[i]);
        }
        return true;
    }
};
```

Note

Insert and search costs **O(key_length)** however, the memory requirements of Trie is `O(ALPHABET_SIZE * key_length * N)` where N is the number of keys in Trie.

There are efficient representations of trie nodes (e.g. compressed trie, ternary search tree, etc.) to minimize the memory requirements of the trie.

How to implement a Trie Data Structure

- Create a root node with the help of **TrieNode()** constructor.
- Store a collection of strings that have to be inserted in the trie in a vector of strings say, **arr**.
- Inserting all strings in Trie with the help of the *insert*() function,
- Search strings with the help of *search*() function.