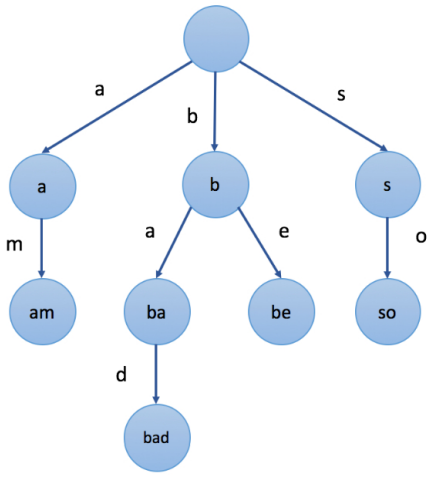
What is Trie?

Report Issue

A `Trie` is a special form of a `Nary tree`. Typically, a trie is used to `store strings`. Each Trie node represents `a string` (`a prefix`). Each node might have several children nodes while the paths to different children nodes represent different characters. And the strings the child nodes represent will be the `origin string` represented by the node itself plus `the character on the path`.

Here is an example of a trie:



In the example, the value we mark in each node is the string the node represents. For instance, we start from the root node and choose the second path 'b', then choose the first child 'a', and choose child 'd', finally we arrived at the node "bad". The value of the node is exactly formed by the letters in the path from the root to the node sequentially.

It is worth noting that the `root` node is associated with the `empty string`.

One important property of Trie is that all the descendants of a node have a common prefix of the string associated with that node. That's why Trie is also called `prefix tree`.

Let's look at the example again. For example, the strings represented by nodes in the subtree rooted at node "b" have a common prefix "b". And vice versa. The strings which have the common prefix "b" are all in the subtree rooted at node "b" while the strings with different prefixes will come to different branches.

Trie is widely used in various applications, such as autocomplete, spell checker, etc. We will introduce the practical applications in later chapters.

How to represent a Trie?

Report Issue

In the previous article, we introduce the concept of Trie. In this article, we will talk about how to represent this data structure in coding languages.

> Briefly review the node structure of a Nary tree before reading the following contents.

What's special about Trie is the corresponding relationship between characters and children nodes. There are a lot of different representations of a trie node. Here we provide two of them.

First Solution - Array

The first solution is to use an `array` to store children nodes.

For instance, if we store strings which only contains letter `a` to `z`, we can declare an array whose size is 26 in each node to store its children nodes. And for a specific character `c`, we can use `c - 'a'` as the index to find the corresponding child node in the array.

C++ Java

Copy

```
// change this value to adapt to different cases
#define N 26

struct TrieNode {
    TrieNode* children[N];

    // you might need some extra values according to different cases
};

/** Usage:
 *  Initialization: TrieNode* root = new TrieNode();
 *  Return a specific child node with char c: (root->children)[c - 'a']
```

It is really `fast` to visit a child node. It is comparatively `easy` to visit a specific child since we can easily transfer a character to an index in most cases. But not all children nodes are needed. So there might be some `waste of space`.

Second Solution - Map

The second solution is to use a `hashmap` to store children nodes.

We can declare a hashmap in each node. The key of the hashmap are characters and the value is the corresponding child node.

C++ Java Copy

```cpp
struct TrieNode {
    unordered_map<char, TrieNode*> children;

    // you might need some extra values according to different cases
};

/** Usage:
 *  Initialization: TrieNode* root = new TrieNode();
 *  Return a specific child node with char c: (root->children)[c]
 */
```

It is even `easier` to visit a specific child directly by the corresponding character. But it might be a little `slower` than using an array. However, it `saves some space` since we only store the children nodes we need. It is also more `flexible` because we are not limited by a fixed length and fixed range.

More

We mentioned how to represent the children nodes in Trie node. Besides, we might need some other values.

For example, as we know, each Trie node represents a string but not all the strings represented by Trie nodes are meaningful. If we only want to store words in a Trie, we might declare a boolean in each node as a flag to indicate if the string represented by this node is a word or not.

```
class TrieNode {

        TrieNode* childs[26];
        bool isEnd;

    public:
          /* Constructor */
            TrieNode(){
                memset(childs, NULL, sizeof(childs));
                isEnd = false;
            }
         /* Helper functions: containsKey() | get() | put() | setEnd() | isEnd()   */
            bool containsKey(char ch){
                return (childs[ch-'a']!=NULL);
            }
            TrieNode* get(char ch){
                return childs[ch-'a'];
            }
            void put(char ch,TrieNode* node){
                childs[ch-'a'] = node;
            }
            void setEnd(){
                isEnd = true;
            }
            bool isEnds(){
                return isEnd;
            }
};

class Trie {
public:
    TrieNode *root;
   /** Initialize your data structure here. */
    Trie() {
        root = new TrieNode();
    }
    /** Inserts a word into the trie. */
    void insert(string word) {
        TrieNode* node  = root;
        for(auto ch : word)
        {
            if(!node->containsKey(ch))
                 node->put(ch,new TrieNode());
            node = node->get(ch);
        }
        node->setEnd();
    }
    /** Returns if the word is in the trie. */
    bool search(string word) {
        TrieNode* node  = root;
        for(auto ch : word)
        {
            if(!node->containsKey(ch))
                return false;
            node = node->get(ch);
        }
        if(node->isEnds())
            return true;
        return false;

    }
    /** Returns if there is any word in the trie that starts with the given prefix. */
    bool startsWith(string prefix) {
        TrieNode* node  = root;
        for(auto ch : prefix)
        {
            if(!node->containsKey(ch))
                return false;
            node = node->get(ch);
        }
        return true;
    }
};

/**
 * Your Trie object will be instantiated and called as such:
 * Trie* obj = new Trie();
 * obj->insert(word);
 * bool param_2 = obj->search(word);
 * bool param_3 = obj->startsWith(prefix);
 */
```