Add and Search Word - Data structure design

Design a data structure that supports the following two operations:

```
void addWord(word)
bool search(word)
```

search(word) can search a literal word or a regular expression string containing only letters `a-z` or `.`. A `.` means it can represent any one letter.

For example:

```
addWord("bad")
addWord("dad")
addWord("mad")
search("pad") -> false
search("bad") -> true
search(".ad") -> true
search("b..") -> true
```

Note:
You may assume that all words are consist of lowercase letters `a-z`.

click to show hint.

You should be familiar with how a Trie works. If not, please work on this problem: Implement Trie (Prefix Tree) first.

Solution 1

This problem is an application of the Trie data structure. In the following, it is assumed that you have solved Implement Trie (Prefix Tree).

Now, let's first look at the `TrieNode` class. I define it as follows.

```
class TrieNode {
public:
    bool isKey;
    TrieNode* children[26];
    TrieNode(): isKey(false) {
        memset(children, NULL, sizeof(TrieNode*) * 26);
    }
};
```

The field `isKey` is to label whether the string comprised of characters starting from `root` to the current node is a key (word that has been added). In this problem, only lower-case letters `a - z` need to be considered, so each `TrieNode` has at most `26` children. I store it in an array of `TrieNode*`: `children[i]` corresponds to letter `'a' + i`. The remaining code defines the constructor of the `TrieNode` class.

Adding a word can be done in the same way as inImplement Trie (Prefix Tree). The basic idea is to create a `TrieNode` corresponding to each letter in the word. When we are done, label the last node to be a key (set `isKey = true`). The code is as follows.

```
void addWord(string word) {
    TrieNode* run = root;
    for (char c : word) {
        if (!(run -> children[c - 'a']))
            run -> children[c - 'a'] = new TrieNode();
        run = run -> children[c - 'a'];
    }
    run -> isKey = true;
}
```

By the way, `root` is defined as private data of `WordDictionary`:

```
private:
    TrieNode* root;
```

And the `WordDictionary` class has a constructor to initialize `root`:

```
WordDictionary() {
    root = new TrieNode();
}
```

Now we are left only with `search`. Let's do it. The basic idea is still the same as typical search operations in a Trie. The critical part is how to deal with the dots `.`.

Well, my solution is very naive in this place. Each time when we reach a `.`, just traverse all the children of the current node and recursively search the remaining substring in `word` starting from that children. So I define a helper function `query` for `search` that takes in a string and a starting node. And the initial call to `query` is like `query(word, root)`.

By the way, I pass a `char*` instead of `string` to `query` and it greatly speeds up the code. So the initial call to `query` is actually `query(word.c_str(), root)`.

Now I put all the codes together below. Hope it to be useful!

```
class TrieNode {
public:
    bool isKey;
    TrieNode* children[26];
    TrieNode(): isKey(false) {
        memset(children, NULL, sizeof(TrieNode*) * 26);
    }
};

class WordDictionary {
public:
    WordDictionary() {
        root = new TrieNode();
    }

    // Adds a word into the data structure.
    void addWord(string word) {
        TrieNode* run = root;
        for (char c : word) {
            if (!(run -> children[c - 'a']))
                run -> children[c - 'a'] = new TrieNode();
            run = run -> children[c - 'a'];
        }
        run -> isKey = true;
    }

    // Returns if the word is in the data structure. A word could
    // contain the dot character '.' to represent any one letter.
    bool search(string word) {
        return query(word.c_str(), root);
    }

private:
    TrieNode* root;

    bool query(const char* word, TrieNode* node) {
        TrieNode* run = node;
        for (int i = 0; word[i]; i++) {
            if (run && word[i] != '.')
                run = run -> children[word[i] - 'a'];
            else if (run && word[i] == '.') {
                TrieNode* tmp = run;
                for (int j = 0; j < 26; j++) {
                    run = tmp -> children[j];
```

```
                    if (query(word + i + 1, run))
                        return true;
                }
            }
            else break;
        }
        return run && run -> isKey;
    }
};

// Your WordDictionary object will be instantiated and called as such:
// WordDictionary wordDictionary;
// wordDictionary.addWord("word");
// wordDictionary.search("pattern");
```

written by jianchao.li.fighter original link here

Solution 2

```
public class WordDictionary {
    WordNode root = new WordNode();
    public void addWord(String word) {
        char chars[] = word.toCharArray();
        addWord(chars, 0, root);
    }

    private void addWord(char[] chars, int index, WordNode parent) {
        char c = chars[index];
        int idx = c-'a';
        WordNode node = parent.children[idx];
        if (node == null){
            node = new WordNode();
            parent.children[idx]=node;
        }
        if (chars.length == index+1){
            node.isLeaf=true;
            return;
        }
        addWord(chars, ++index, node);
    }


    public boolean search(String word) {
        return search(word.toCharArray(), 0, root);
    }

    private boolean search(char[] chars, int index, WordNode parent){
        if (index == chars.length){
            if (parent.isLeaf){
                return true;
            }
            return false;
        }
        WordNode[] childNodes = parent.children;
        char c = chars[index];
        if (c == '.'){
            for (int i=0;i<childNodes.length;i++){
                WordNode n = childNodes[i];
                if (n !=null){
                    boolean b = search(chars, index+1, n);
                    if (b){
                        return true;
                    }
                }
            }
            return false;
        }
        WordNode node = childNodes[c-'a'];
        if (node == null){
            return false;
        }
        return search(chars, ++index, node);
    }
```

```
    private class WordNode{
        boolean isLeaf;
        WordNode[] children = new WordNode[26];
    }
}
```

written by qgambit2 original link here

Solution 3

Using backtrack to check each character of word to search.

```
public class WordDictionary {
    public class TrieNode {
        public TrieNode[] children = new TrieNode[26];
        public String item = "";
    }

    private TrieNode root = new TrieNode();

    public void addWord(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            if (node.children[c - 'a'] == null) {
                node.children[c - 'a'] = new TrieNode();
            }
            node = node.children[c - 'a'];
        }
        node.item = word;
    }

    public boolean search(String word) {
        return match(word.toCharArray(), 0, root);
    }

    private boolean match(char[] chs, int k, TrieNode node) {
        if (k == chs.length) return !node.item.equals("");
        if (chs[k] != '.') {
            return node.children[chs[k] - 'a'] != null && match(chs, k + 1, node.
children[chs[k] - 'a']);
        } else {
            for (int i = 0; i < node.children.length; i++) {
                if (node.children[i] != null) {
                    if (match(chs, k + 1, node.children[i])) {
                        return true;
                    }
                }
            }
        }
        return false;
    }
}
```

written by Lnic original link here