Trie Data Structure - Explained with Examples

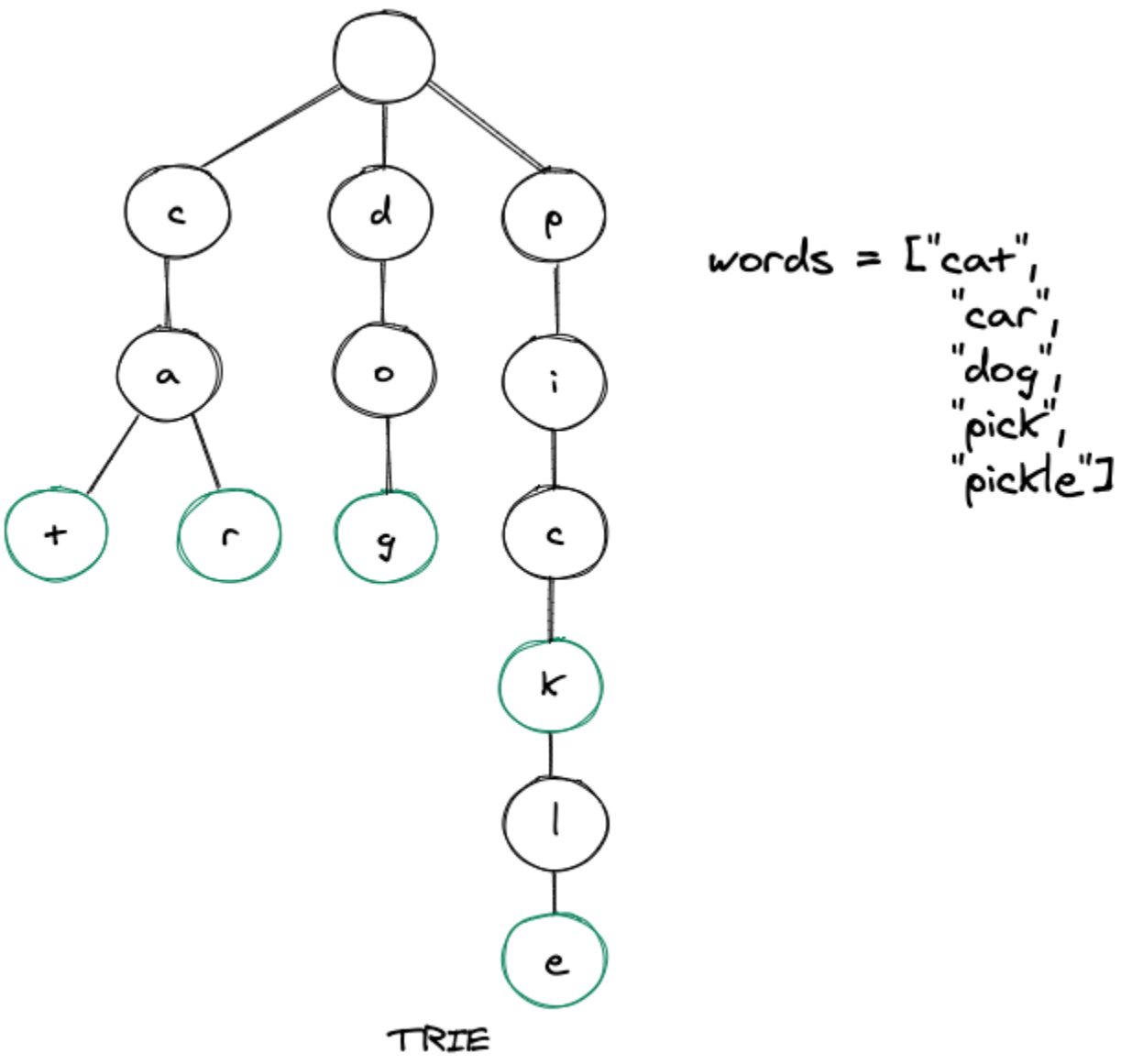
A **Trie is an advanced data structure** that is sometimes also known as **prefix tree or digital tree**. It is a tree that stores the data in an ordered and efficient way. We generally **use trie's to store strings**. Each node of a trie can have as many as 26 references (pointers).

Each node of a trie consists of two things:

- A character
- A boolean value is used to implement whether this character represents the end of the word.

Tries in general are used to store English characters, hence each character can have 26 references. Nodes in a trie do not store entire keys, instead, they store a part of the key(usually a character of the string). When we traverse down from the root node to the leaf node, we can build the key from these small parts of the key.

Let's build a trie by inserting some words in it. Below is a pictorial representation of the same, we have 5 words, and then we are inserting these words one by one in our trie.



As it can be seen in the image above, the key(words) can be formed as we traverse down from the root node to the leaf nodes. It can be noted that the green highlighted nodes, represents the `endOfWord` boolean value of a word which in turn means that this particular word is completed at this node. Also, the root node of a trie is empty so that it can refer to all the members of the alphabet the trie is using to store, and the children nodes of any node of a trie can have at most 26 references. Tries are not balanced in nature, unlike AVL trees.

Why use Trie Data Structure?

When we talk about the **fastest ways to retrieve values** from a data structure, **hash tables generally comes to our mind**. Though very efficient in nature but still very less talked about as when compared to hash tables, **trie's are much more efficient than hash tables** and also they possess several advantages over the same. Mainly:

- There won't be any collisions hence making the worst performance better than a hash table that is not implemented properly.
- No need for hash functions.
- Lookup time for a string in trie is `O(k)` where **k = length of the word**.
- It can take even less than `O(k)` time when the word is not there in a trie.

Implementing Tries:

We will implement the Trie data structure in Java language.

Trie Node Declaration:

```
class TrieNode {

    boolean isEndOfWord;

    TrieNode children[];



    public TrieNode(){

        isEndOfWord = false;

        children = new TrieNode[26];

    }

}
```

Copy
Note that we have two fields in the above **TrieNode** class as explained earlier, the boolean isEndOfWord keyword and an array of Trie nodes named children. Now let's initialize the root node of the trie class.

```
TrieNode root;



public Trie() {

    root = new TrieNode();

}
```

Copy
There are two key functions in a trie data structure, these are:

- **Search**
- **Insert**

Insert in trie:

When we insert a character(part of a key) into a trie, we start from the root node and then search for a reference, which corresponds to the first key character of the string whose character we are trying to insert in the trie. Two scenarios are possible:

- A reference exists, if, so then we traverse down the tree following the reference to the next children level.
- A reference does not exist, then we create a new node and refer it with parents reference matching the current key character. We repeat this step until we get to the last character of the key, then we mark the current node as an end node and the algorithm finishes.

Consider the code snippet below:

```
public void insert(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.isEndOfWord = true;

}
```

Copy

Search in trie:

A key in a trie is stored as a path that starts from the root node and it might go all the way to the leaf node or some intermediate node. If we want to search a key in a trie, we start with the root node and then traverses downwards if we get a reference match for the next character of the key we are searching, then there are two cases:

1. A reference of the next character exists, hence we move downwards following this link, and proceed to search for the next key character.
2. A reference does not exist for the next character. If there are no more characters of the key present and this character is marked as `isEndOfWord = true`, then we return **true**, implying that we have found the key. Otherwise, two more cases are possible, and in each of them we return **false**. These are:
    1. There are key characters left in the key, but we cannot traverse down as the path is terminated, hence the key doesn't exist.
    2. No characters in the key are left, but the last character is not marked as `isEndOfWord = false`. Therefore, the search key is just the prefix of the key we are trying to search in the trie.

Consider the code snippet below:

```
public boolean search(String word) {

    return isMatch(word, root, 0, true);

}



public boolean startsWith(String prefix) {

    return isMatch(prefix, root, 0, false);

}



public boolean isMatch( String s, TrieNode node, int index, boolean isFullMatch) {

    if (node == null)

        return false;



    if (index == s.length())

        return !isFullMatch || node.isEndOfWord;



    return isMatch(s, node.children[s.charAt(index) - 'a'], index + 1, isFullMatch);

}
```

Copy
The method `startsWith()` is used to find if the desired key prefix is present in the trie or not. Also, both the `search()` and `startsWith()` methods make use of `isMatch()` method.

Entire Code:

```
class Trie {

    class TrieNode {
```

```
        boolean isEndOfWord;
        TrieNode children[];

        public TrieNode(){
            isEndOfWord = false;
            children = new TrieNode[26];
        }
    }

    TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(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.isEndOfWord = true;
    }

    public boolean search(String word) {
        return isMatch(word, root, 0, true);
    }

    public boolean startsWith(String prefix) {
        return isMatch(prefix, root, 0, false);
```

```
    }


    public boolean isMatch( String s, TrieNode node, int index, boolean isFullMatch) {
        if (node == null)
            return false;

        if (index == s.length())
            return !isFullMatch || node.isEndOfWord;

        return isMatch(s, node.children[s.charAt(index) - 'a'], index + 1, isFullMatch);
    }


    public static void main(String[] args){
        Trie trie = new Trie();
        trie.insert("cat");
        trie.insert("car");
        trie.insert("dog");
        trie.insert("pick");
        trie.insert("pickle");
        boolean isPresent = trie.search("cat");
        System.out.println(isPresent);
        isPresent = trie.search("picky");
        System.out.println(isPresent);
        isPresent = trie.startsWith("ca");
        System.out.println(isPresent);
        isPresent = trie.startsWith("pen");
        System.out.println(isPresent);


    }
}
```

Copy
The output of the above looks like this:

true

false

true

false

Trie Applications

- The most common use of tries in real world is the **autocomplete feature** that we get on a search engine(now everywhere else too). After we type something in the search bar, the tree of the potential words that we might enter is greatly reduced, which in turn allows the program to enumerate what kinds of strings are possible for the words we have typed in.
- Trie also helps in the case where we want to store additional information of a word, say the popularity of the word, which makes it so powerful. You might have seen that when you type **"foot"** on the search bar, you get **"football"** before anything say **"footpath"**. It is because **"football"** is a much popular word.
- Trie also has helped in checking the correct spellings of a word, as the path is similar for a slightly misspelled word.
- **String matching** is another case where tries to excel a lot.

Key Points

- The time complexity of creating a trie is `O(m*n)` where m = number of words in a trie and n = average length of each word.
- Inserting a node in a trie has a time complexity of `O(n)` where n = length of the word we are trying to insert.
- Inserting a node in a trie has a space complexity of `O(n)` where n = length of the word we are trying to insert.
- Time complexity for searching a key(word) in a trie is `O(n)` where n = length of the word we are searching.
- Space complexity for searching a key(word) in a trie is `O(1)`.
- Searching for a prefix of a key(word) also has a time complexity of `O(n)` and space complexity of `O(1)`.

Conclusions

- We learned what a Trie is, and why do we need one.
- We also implemented trie in Java, where insert and search were the main methods implemented.
- We then talked about the applications of Trie data structure in real world, followed by several key points that we should also remember about trie's.
