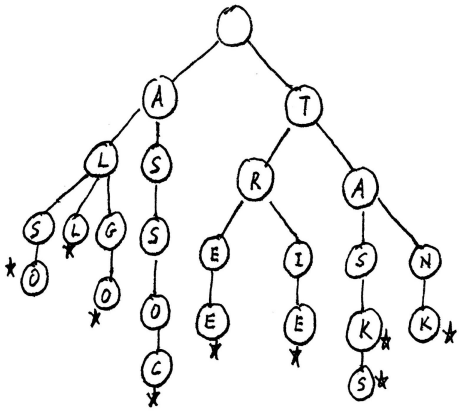
Trie

Definition

A trie, or prefix tree, is a special data structure that functions as a collection of strings (or any type of sequence where the absolute location in the sequence matters). Before we dive into the applications and advantages of using this data structure over others, let us first see what it is. (from this point on, I will talk about trie in the context of storing a list of strings, but similar concept can be applied to all type of sequential data structures)

A trie is a tree where each node holds one character. Each vertex is an **associative prefix** to each of its child nodes. The tree is rooted at the empty string, which is the universal prefix of the dictionary.

Pictorially a dictionary containing "also," "all," "algo," "assoc," "tree," "trie," "task," "tasks," "tank" looks like:



Note that the end of words are marked by a star, and note that "task" is completed contained in the branch of "tasks." This highlights one advantage of using tries (not an important one, but pretty neat) which is that it offers some degree of compression for a dictionary of strings where a lot of strings are similar.

Note that another thing tries are amazing at and commonly used for is that it can query which strings in a dictionary have a common prefix, as if they have a common prefix, that prefix must be their common ancestor in the tree.

Operations and Implementation

Node

A node fundamentally need only 2 things, how many words terminate on this node (can be replaced by a boolean if repeated words are not allowed), and a list of references to its children. If this trie is used in a prefix-query heavy context, one can also store how many words have the current node as a common prefix. In code the structure should have the following fields:

```
class TrieNode {
    int wordCount;
    int prefixCount;
    Map<Character, TrieNode> children;
}
```

Insert

Insertion can be repeated until the string becomes the empty string:

```
    void insert(String s) {
        if (s.length == 0) {
            wordCount++;
            prefixCount++;
            return;
        }

        char head = s.charAt(0);

        if (!children.containsKey(head)) {
            children.put(head, new TrieNode(0, 0, new HashMap<Character,
   TrieNode>()));
        }

        children.get(head).insert(s.substring(1));

        prefixCount++;
    }
```

This operation have runtime $O(L)$ where L is the length of the string.

Contains

To check if a word is in the dictionary we similarly repeatedly ask until `s` becomes an empty string:

```
    boolean contains(String s) {
        if (s.length == 0) {
            return wordCount > 0; // else this is not a word
```

```
        }

        char head = s.charAt(0);

        if (!children.containsKey(head)) {
            return false;
        }

        return children.get(head).contains(s.substring(1));
    }
```

This operation have runtime $O(L)$ where L is the length of the string. (Note that this is $O(1)$ in terms of size of the trie)

Query Prefix

To check how many words have a given prefix we simply advance to the node representing the prefix and return its `prefixCount`:

```
    int countPrefix(String prefix) {
        if (prefix.length == 0) {
            return prefixCount;
        }

        head = prefix.charAt(0);

        if (!children.containsKey(head)) {
            return 0; // prefix doesn't exist
        }

        return children.get(head).countPrefix(prefix.substring(1));
    }
```

This operation have runtime $O(L)$ where L is the length of the prefix.

Variations

There are a lot of modifications we can make to the trie data structure to make it more versatile. To start we can turn trie into an **associative data structure**, more commonly known as a map. If we were to turn it into a map we would just need an extra field within each node to store its `content`, which may be null if this is not a key in the map. This offers $O(L)$ access time where L is the length of the key.

Tries can also be used on primitive data types in the form of a **bitwise-trie**. It takes the primitive data type and treats them as a bit-array. This produces effectively a cache-independent binary search tree that is in fact highly optimized and parallelized.

Further, the implementation went over here uses recursion for most of the operations. While this is more intuitive than the iterative approach it is slower due to how imperative pro-

gramming languages function. I will send out a version of trie that uses iteration instead of recursion with the same operations as well.