Tries

A trie is a type of tree which efficiently stores a dictionary of words. An idea usually shown to first semester students that a trie uses is the idea of a frequency array. In a frequency array of characters freq[0] stores the number of a's seen, freq[1] stores the number of b's seen, and so forth. The salient part of this type of storage is that the character is implicit in the index. We never actually store 'a', 'b', or any of the letters. Rather, we exploit the ordering of the letters, and assign a meaning based on the index within which something is stored. A trie exploits this same exact idea of implicitly storing a letter via the index into an array.

A trie is a tree, where each node represents a letter in the prefix of some word stored in a dictionary of words. The most basic trie node has the following components:

```
trienode* next[26]; // An array of 26 pointers.
int flag; // flag=1 if this node is a word, 0 otherwise.
```

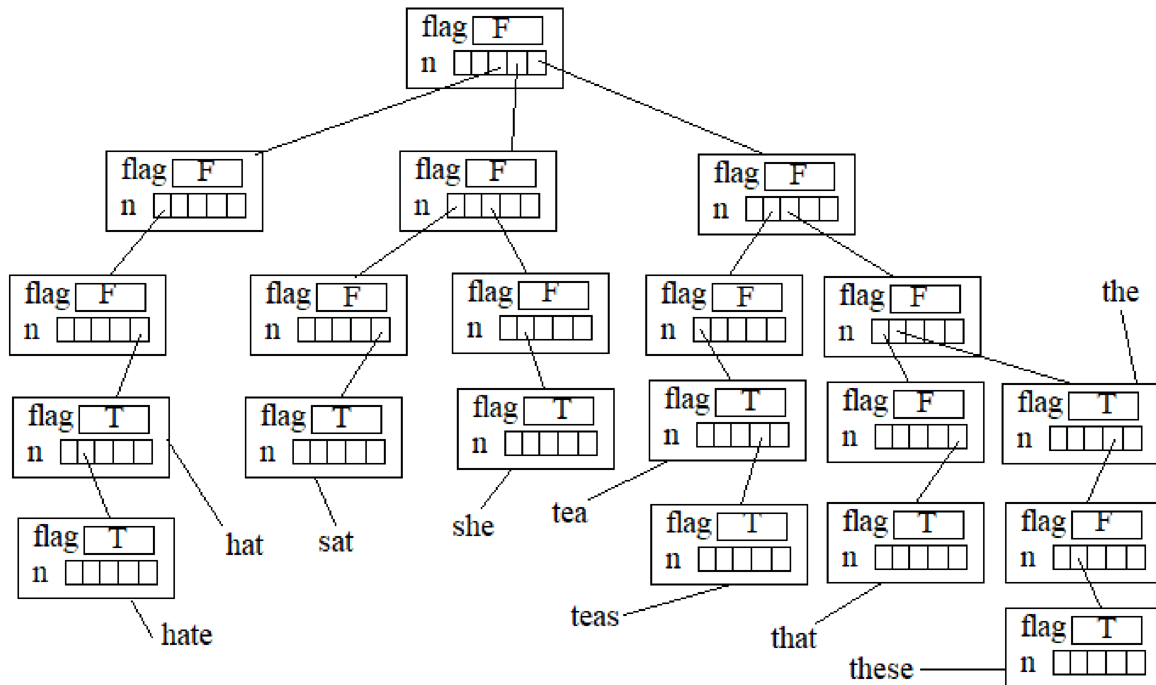
The flag will store true if this prefix is actually a word, and false if it is not. The pointer next[i] will be NULL if there is no word in the dictionary that starts with the given prefix followed by the letter i represents. Otherwise, the pointer would point to a node representing the given prefix followed by the letter i represents.

To visualize a trie, let's use a reduced alphabet with the letters A, E, H, S, and T assigning these to the numbers 0, 1, 2, 3 and 4, respectively. Let our trie store the following words:

HAT, HATE, SAT, SHE, TEA, TEAS, THAT, THE and THESE

Note that the root node of a trie represents the empty prefix.

A picture of the trie which stores these words is included on the next page. In order to make the picture fit nicely, 'T' is use to represent true and 'F' is used to represent false, and a lowercase n is used to represent the array next. The nodes which represent each prefix that correspond to a word are also indicated. Notice that the key is that which links are active is how all the different words are stored. Thus, all we need to do to store a word is just store "true" in the node that represents that prefix. Also, in the picture, instead of writing 0 for false, 'F' is written and instead of writing 1 for true, 'T' is written. (Note: I originally write these notes for Java where there is a Boolean type. I didn't think it was worth the time to redraw the entire drawing just to change the Ts and Fs to 1s and 0s, respectively.)



Inserting a node in a trie

If we do this iteratively, we can simply start at the root, and then follow the links corresponding to the appropriate letters in the word. If a link doesn't exist, we must create that node (and each of the subsequent ones.) When we get to the last node, we must simply change its flag value to 1, to indicate that the word to be inserted ends there.

Recursively, we can pass in the node the insertion is being done on, the word, and an integer k, indicating how many letters have been read in so far. If we use this system, our base case is when k is the length of the word. If this is the case, we just set flag to 1 and return. Otherwise, we recursively insert into the next node. If the next node doesn't exist, we create it first. Here is C code for trie insertion:

```
typedef struct TrieNode {
    struct TrieNode *children[26];
    int flag; // 1 if the string is in the trie, 0 otherwise
} TrieNode;


void insert(TrieNode* tree, char word[], int k) {

    if (k == strlen(word)) {
        tree->flag = 1;
        return;
    }

    int nextIndex = word[k] - 'a';
    if (tree->children[nextIndex] == NULL)
        tree->children[nextIndex] = init();
    insert(tree->children[nextIndex], word, k+1);
}
```

Searching for a node in a trie

Searching is fairly similar to inserting. Just follow the path using the letters in the word, either iteratively or recursively. They key difference is that if a next link is NULL, instead of creating a node, that is proof that the word being searched for is NOT stored in the trie, so you just stop and return 0.

Insert/Search Run Times

It should come as no surprise that the run-times for both inserting and searching for a word in a trie are O(len), where len is the length of the word in question. Thus, tries are ideal for real dictionaries of words, where no individual word is too long, but there are many words to search from. The maximum storage of a trie is on the order of the total number of letters of all of the words in the dictionary, but may be less than this due to all shared prefix letters.

Storing Extra Information in a Trie Node

It's typical that in order to solve a problem, the basic trie node must be edited, to store extra information. One simple piece of information would be an extra integer representing the number of words stored in that particular subtree. When we add a field to store some piece of information in each node, we must then update that piece of information in every node. For this specific example, when we insert a word, we must add 1 to the numwords field of each node ancestor node of the last node of the word. The only thing that would change of the insert shown on the previous page is adding this line to it first:

```
    tree->numwords += 1;
```

Now, imagine having to answer queries of the form, "How many words start with the prefix 'trans'?" To answer this question, we just search to the node corresponding to the prefix "trans", and then just return what's stored in the numwords field of that node.

Solving a Problem: What word has the most prefixes in it that are ALSO words?

Consider the word "intention". Its prefixes, "i", "in", "intent" and "intention", are all words. In general, consider solving the problem: given a trie storing a dictionary of words, find which word has the most prefixes which are also words. The answer to this question is the SAME as finding the path in a trie from the root to a leaf which go through the most number of nodes storing 1 for their flag field. Here is some C code which solves this problem:

```
int maxNumPrefixWords(TrieNode* root) {

    if (root == NULL) return 0;
    int maxChild = 0;
    int i;
    for (i=0; i<26; i++)
        maxChild = max(maxChild, maxNumPrefixWords(root->children[i]));

    return maxChild + root->flag;
}
```

Basically, we want the best path of all of our subtrees. Then, we want to take this value and add our root node value to this number, since our root node adds to any of the subtree paths.

<u>Alien Rhyme Problem (Google Code Jam 2019 Round 1A)</u>
In this problem, you are given upto a thousand words, with a max length of 50 and would like to pair up as many of them as possible such that the suffix in each pair is unique and no other word in the group has the same suffix.

Although it might not seem like, it, we can use the same exact ideas as those previously mentioned to solve the problem.

First, notice that suffixes of words are stored in different places in a trie, so this storage system would be unnatural to help pair up words. But…consider the idea of reversing each word. Then, suffixes become prefixes (in reversed order). Furthermore, if we just keep a numWords field in each trie node, and if this field stores the number 2, this means that exactly 2 words have this suffix and we should pair them up! So, to solve the problem – read in all of the words in reverse order, storing the number of words in each subtrie. This data structure is now nicely set up to solve the problem at hand.

So, we'll do the following:

1. Recursively solve the problem for each sub-node. We can definitely add up all of these answers, since no two sub-nodes will store the same suffix.

But, this will miss a few corner cases. If in doing this, there are at least two words that weren't paired up, we are allowed to pair up these two words by then making the root node the accented node (since it's not accented for any of the other words.) If there are more than 2 words that weren't picked it's okay, since we're allowed to discard some words. But, we can just keep 2 of them.

Here is C code that solves the key part of the problem:

```
// Solves the problem the node root which is at depth depth.
int solve(TrieNode* root , int depth) {

        // We can grab both...
        if (root->numWords == 2 && depth > 0) return 2;

        // Try adding all the words from subtrees.
        int res = 0;
        for (int i=0; i<26; i++) {
                if (root->next[i] != NULL)
                        res += solve(root->next[i], depth+1);
        }

        if (root->numWords-res >= 2 && depth != 0) res += 2;

        return res;
}
```

Scrabble Problem (past Foundation Exam)

In scrabble, you are given a set of tiles with letters and attempt to form words with those tiles. Consider the problem of counting the number of unique words that can be formed with the letters, assuming that the dictionary of valid words is stored in a trie and the tile information is given to you in a frequency array. (For example, freq[4] would store the number of 'E' tiles you have.) Here is the function prototype, in C:

```
int countWords(TrieNode* root, int freq[]);
```

The function should return the total number of words that can be formed in the trie with the root root, using the letters in freq.

The idea is as follows:

If the node itself is a word, we add 1 to our count. This is a base case of sorts.

Next, we see what letter to place next. We try all 26 letters. But, we skip a letter if either

(a) we don't have that tile OR
(b) there are no dictionary words that have that letter next.

If both conditions are met (we have the tile AND there is some dictionary word down this path), we go ahead and "play" the tile by subtracting 1 from the appropriate place in the frequency array, and then recursing.

For each of these upto 26 recursive calls we make, add up the results, since each of these subtries have separate words we may want to count, and then return the answer!

Here is the code:

```
int countWords(TrieNode* root, int freq[]) {

    int res = root->flag; // 0 or 1 which is appropriate.

    for (int i=0; i<26; i++)
        if (root->next[i] != NULL && freq[i] > 0) {
            freq[i]--;
            res += countWords(root->next[i], freq);
            freq[i]++;
        }
    }
    return res;
}
```