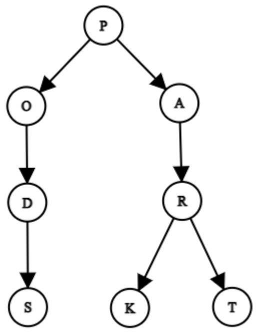
Data Structure: Trie

A Trie, sometimes called a Prefix Tree, is a tree data structure used to locate keys in a set. Tries are most commonly used to store a set of strings, with each node being connected by an individual letter. To access a key, the Trie is traversed depth-first, character by character.

Applications of Tries include predictive texts for autocompleting, spell checking, and hyphenation software. Tries are used in these applications because keys can be quickly inserted, deleted, and searched for.



A Trie containing the words Pod, Pods, Par, Park, and Part.

Implementation

Tries can be implemented using nodes and pointers to form a tree. This allows for easy insertion and deletion. Each node needs to store pointers to its children, a pointer to its parent, a flag stating if it ends a word, and the character of the node.

```
struct trie_node {
    map<char, trie_node*> children;
    trie_node* parent;
    bool end = false;
    char letter;
};
```

Insertion

A word can be inserted into a Trie in **O(m)** time, and increases the memory usage up to **O(m)**, where **m** is the length of the word. To insert a word, the tree is traversed letter by letter to match the word being inserted. If there a matching child node does not exist, new nodes are added to the Trie. When all the letters for the word are in the Trie, the final node is flagged as an ending node, adding the word to the structure.

```cpp
void insert(trie_node* head, string word) {
    trie_node* node = head;
    // add the word letter by letter
    for (char c : word) {
        trie_node* next = node->children[c];
        if (next == NULL) {
            // this letter is not in the trie
            // create a new node
            trie_node* new_node = new trie_node();
            new_node->parent = node;
            new_node->letter = c;
            node->children[c] = new_node;
            // move to this node
            node = new_node;
        }
        else node = next;
    }
    // mark that this node ends a word
    node->end = true;
}
```

Searching

To search for a word in a Trie, the tree is traversed letter by letter to match the word being searched for. If no such child node exists, the word is not in the Trie. Once all the letters in the word have been found, the last node is checked to see if a word ends there. If one does, the word in the Trie, else it is not. As with insertion, searching is done in **O(m)** time, where **m** is the length of the word.

```
bool contains(trie_node* head, string word) {
    trie_node* node = head;
    // check for presence letter by letter
    for (char c : word) {
        trie_node* next = node->children[c];
        if (next == NULL) return false;
        node = next;
    }
    // does a word finish here?
    return node->end;
}
```

Deletion

There are two ways to delete a word from a Trie. The simpler method is to find the end of the word and remove the ending flag from the last node. However, this does not free up memory and leaves unused nodes in the tree. The second method is similar to the first, but after the ending flag is remove, the tree is traversed back up to the top, removing any nodes which have no children and are not flagged as ending nodes. Both methods take **O(m)** time, and the second method releases up to **O(m)** memory, where **m** is the length of the word.

```
void erase(trie_node* head, string word) {
    trie_node* node = head;
    // find the end of the word letter by letter
    for (char c : word) {
        trie_node* next = node->children[c];
        if (next == NULL) return; // word is not in trie
        node = next;
    }
    // word no longer ends here
    node->end = false;
    // work back up and delete unnecessary nodes
    while (node != head) {
        if (node->children.size() == 0 && !node->end) {
            // node can be deleted
            trie_node* parent = node->parent;
            parent->children.erase(node->letter);
            delete node;
            node = parent;
        }
        else node = node->parent;
    }
}
```

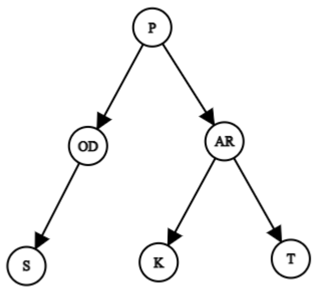
Finding Prefixes

A common use of Tries is to find all words that start with a given prefix. To achieve this, the end of the prefix is found and the remaining subtree is exhaustively searched depth-first to construct all of the words. This takes **O(q+m)** time, where **q** is the total number of nodes in the subtree and **m** is the length of the prefix.

```
vector<string> find_prefix(trie_node* head, string prefix) {
    trie_node* node = head;
    vector<string> words;
    // find the node at the end of the prefix
    for (char c : prefix) {
        trie_node* next = node->children[c];
        if (next == NULL) return words; // prefix is not in trie
        node = next;
    }
    // dfs to find all words
    deque<pair<string, trie_node*>> q;
    q.push_front(make_pair(prefix, node));
    while (!q.empty()) {
        // retrieve current node
        pair<string, trie_node*> p = q.front();
        q.pop_front();
        string str = p.first;
        trie_node* node = p.second;
        if (node->end) words.push_back(str); // add word to vector
        // dfs on children
        for (auto element = node->children.begin(); element !=
        ↪   node->children.end(); element++)
            if (element->second != NULL)
                q.push_front(make_pair(str + element->first,
                ↪   element->second));
    }
    // return all words found
    return words;
}
```

Optimisations

Tries can be memory intensive as each letter gets its own node. The size of each node can be reduced by removing the *parent* and *letter* variables and altering the deletion function. If a Trie contains many words with few overlapping characters, a Radix Tree may be useful. To reduce memory usage, Radix Trees compress nodes without siblings with their parents. However, this results in slighlty slower times for insertion, deletion and searching.



A Radix Tree containing the words Pod, Pods, Par, Park, and Part.