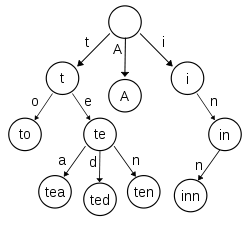
**Trie Implementation:**

**Part - Background Theory**: A trie is a data structure typically used to store and look up words. You have a trie node, which contains a char variable called character, a boolean variable called word complete, and a hashmap (of character, trie node) called children.



Suppose someone wanted to look up “ten” with the trie above. They would need the root trie node (the one with an empty character). Then they would look up “ten” by looking for a child with a character “T”, then a child with a character “e”, then check if that child has a child with a character “n”. The leaves of the tree are all the words, all other nodes are just “paths” and serve to route searches with results. In the diagram, the center of the leaf reads “ten” but in reality its character is ‘n’. You know it is a word because of two components. The “word complete” boolean would be set to true which would indicate that it is a complete word and not a leaf due to other reasons like incomplete word additions. Then the path to get there from the root would describe the word.

**Part 2 – Trie Node Implementation:**

|  |
| --- |
| **public** **class** TrieNode {  **private** HashMap<Character, TrieNode> children;  **private** **boolean** wordComplete = **false**;  **private** **char** character;  **public** TrieNode() {  children = **new** HashMap<Character, TrieNode>();  }    **public** TrieNode(**char** character) {  **this**();  **this**.character = character;  }  **public** **char** getChar() {  **return** character;  }  **public** TrieNode getChild(**char** c) {  **return** children.get(c);  }  **public** **boolean** wordComplete() {  **return** wordComplete;  }    **public** **void** setWordComplete() {  wordComplete = **true**;  }  **public** **void** addWord(String word) {  **if** (word == **null** || word.isEmpty()) {  **return**; //don't bother adding empty words  }    //Get the first char and ensure it exists in the trie  **char** firstChar = word.charAt(0);  TrieNode child = getChild(firstChar);  **if** (child == **null**) {  child = **new** TrieNode(firstChar);  children.put(firstChar, child);  }  //Ensure the rest of the word exists in the trie one char at a time  **if** (word.length() > 1) {  child.addWord(word.substring(1));  } **else** {  child.setWordComplete();//set this to true for the last char  }  }  } |

**Part 3 – Trie Implementation:**

|  |
| --- |
| **public** **class** Trie{  **private** TrieNode root;  **public** Trie(List<String> list) {  root = **new** TrieNode();  **for** (String word : list) {  root.addWord(word);  }  }  **public** Trie(String[] list) {  root = **new** TrieNode();  **for** (String word : list) {  root.addWord(word);  }  }  **public** TrieNode getRoot() {  **return** root;  }    **public** **boolean** contains(String prefix) {  TrieNode lastNode = root;  **for** (**int** i = 0; i < prefix.length(); i++) {  //Keep trying to iterate down the trie one char at a time  lastNode = lastNode.getChild(prefix.charAt(i));  **if** (lastNode == **null**) {  **return** **false**; //If the child is null the path doesn’t exist  }  }  **return** **true**; //If it reaches this point, the word is in the list  }  } |

**Summary:** The two constructors just initialize a trie that includes all the words in the string collection (array or list). The contains method searches for any prefixes. It could be used to search for words also.

**Part 4 – Trie Time Complexities:**

There are really two functions that affect time complexity. That is the “add word” (often called insert) and the “contains” (which is often referred to as the search).Also, there is the constructor to initialize a try by calling a “add words” a number of times.

**“Add Word”:** O(n) where n is the number of characters in the string.

For each character in the string, you call “char at” once, you call “hashmap get” once, and then you call “substring” once. The other things that are done are either done once (on the last char) or are constant and don’t matter. So the Big Oh is

O( C\* n \* (get + charAt + substring))If you look it up, the string “char at”, hashmap “get”, and “substring”, there all constant. So the time complexity for “add word” is O(n).

**“Contains”:** O(n) where n is the number of characters in the string.

For each character in the string, you call a hashmap “get” and a string “char at” which are both constant. The for loop is why it is O(n).

**Constructors:** O(mn) where m is the number of words in the collection and n is the number of characters in the string.

It just calls “add words” O(n) m times.

**IMPORTANT NOTE:** As of Java 7 update 6, the substring method has been changed so that it has an O(n) complexity. This would degrade your performance of the trie. You could optimize this by re-writing your own O(1) substring method or looking up the old code in the JDK open source for previous versions when it was O(1).

**Part 5 – Space Complexity:**