# Assignment 5

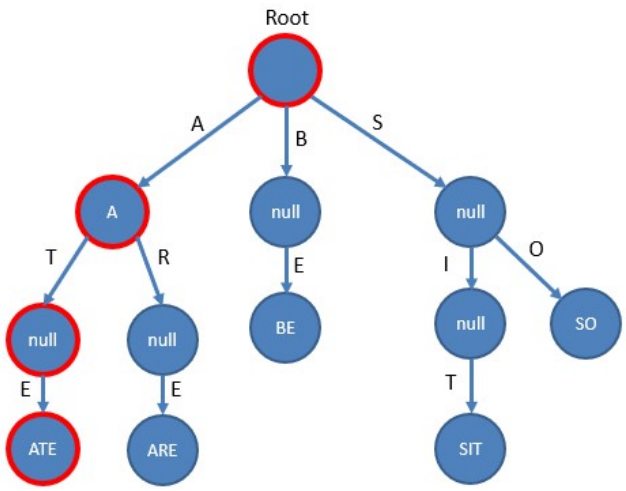
## Recursion

---

Submit a single ZIP file called **assignment5.zip** containing each of your Java files. **Your zip file must contain only .java files organized into a single folder (i.e., no packages, sub-folders, etc.) and the TAs should be able to compile all your code within this same folder (i.e., your code should also not reference packages).**

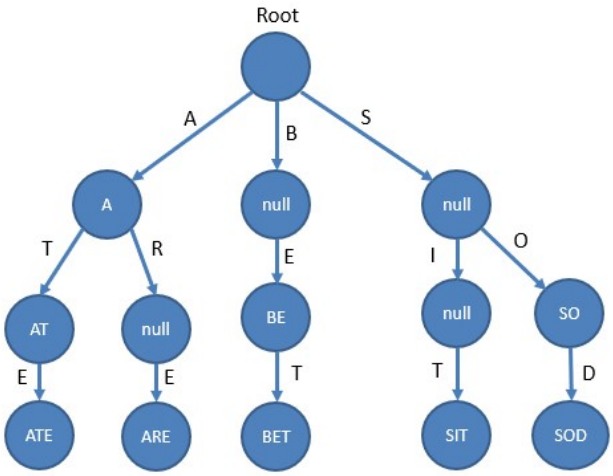
---

A trie, also known as a prefix tree, is a tree-based data structure that stores key/value pairs (like a HashMap). The keys in tries are usually strings. In this assignment, you will implement a trie data structure that uses recursion to store string keys and their associated string values. The root node in the trie will be empty (no key or value). Each node in the trie, including the root, will store a hash map with character keys and trie node values. This structure will allow you to search for a string key in the trie by repeatedly looking up the node associated with the next character in the key (if it exists). As an example, consider the picture below (ignore the red for now). This trie contains the keys/values “A”, “ATE”, “ARE”, “BE”, “SIT”, and “SO” (assume the key and value are the same, null values indicate nothing has been added there).




If you were to look up the key “ATE” in the trie, you would end up following the red node path. Starting at the root node, look up and go to the node associated with ‘A’. From that node, look up and go to the node associated with ‘T’, and then look up and go to the node associated with ‘E’. If the value stored in this final node is null, the key does not exist in the trie. If the value is not-null, the node’s value represents the value associated with the given key (alternatively, if you are performing a put operation, you can add the value to

the node). If you were to look up the key “BAT”, you would start at the root node, look up and go to the node associated with ‘B’, and then determine that they key “BAT” is not in the trie since ‘A’ is not associated with the current node (i.e., is not in that node’s hashmap). If the additional keys “AT”, “BE”, and “SOD” were added, the trie would then look like:




An interesting property of the trie structure is that the key for any node contained within a subtree rooted at a node *X* has the key that would lead to *X* as a prefix. As examples, “SIT”, “SO”, and “SOD” all begin with the character ‘S’ and are all contained within the sub-trie rooted at the node you would reach if you looked up the key ‘S’. If “SOME”, “SODA”, and “SOAR” were added to the above trie, they would all be contained in the subtrees of the key “S” and the key “SO”. This allows tries to easily produce a set of keys that begin with a particular prefix.

To start the assignment, download the Assignment 5 – Base Code.zip file from cuLearn. This zip contains 4 Java files:

1. TrieMapInterface: This interface defines the methods that your TrieMap class must support. These methods should work similarly to how they would for a hash map or any other map data structure. The **put** method should add the associated key/value pair to the trie. If the key is already in the trie, the value should be updated. The **get** method should return the value associated with the given key, if the key exists. If the key does not exist in the trie, the **get** method should return null. The **containsKey** method should return true if the trie contains the given key and false otherwise. The **getValuesForPrefix** method must return an ArrayList of String values that contains all keys within the trie that start with the specified prefix. The **print** method should print all of the values contained within the trie.
2. TrieMap: This class contains skeleton code outlining the methods required to implement the TrieMapInterface. I have also left method signatures from my own solution, which may provide hints regarding possible solutions. You can remove or otherwise change the class however you

wish, so long as your class still implements the TrieMapInterface. **All method implementations must function recursively. This is the only class that you must add code to in order to complete the assignment. You are free to add code to the other classes if you wish.**

3. TrieMapNode: The class representing a node in the trie. It contains a hashmap with character keys and TrieMapNode values (similar to a binary tree but with more than left/right children), along with a constructor and necessary get/set methods. You don't need to change this class but can make changes if you want.
4. TrieMapTester: This class will run several tests on your TrieMap implementation and count the number of errors. This will run a significant number of operations, so it is probably a good idea to run your own smaller tests first to verify your TrieMap implementation is likely correct.

The implementation of the TrieMap class may seem difficult initially. However, after becoming familiar with recursively moving through the trie structure, you will find that most methods are quite similar. So completing the first few methods is likely to be significantly more difficult than completing the others. Personally, I would recommend implementing and testing the methods in this order: constructor, put, print, containsKey, get, getValuesForPrefix.

## Grade Breakdown

Put Method: 10 marks
Get Method: 5 marks
Contains Key Method: 5 marks
Get Values for Prefix Method: 20 marks
Print Method: 10 marks
Total: 50 marks