ECE 242 Project 4

For this project, we will be dealing with Hash Tables and Trees. Using a dictionary file that we provide, you will hash all the words from the file and create a Hash Table. Once you have a fully-populated Hash Table, you will create a little 'game' that will involve using the Hash Table and a Tree. All will be revealed shortly.

Hash Table

In this project, the Hash Table will use a chaining mechanism. What this means is that the table will essentially be an array of Linked Lists.

Step 1 – Creating the hash table objects:

Create a class called '**ListNode**.' This object functions similar to the Linked List object nodes that you've seen countless times throughout the course of the semester. These objects will store a single dictionary word listing and an object reference to the next ListNode in the Linked List.

Step 2 – Creating the hash table:

Create an array of ListNode objects to store each of the Linked Lists that you create. Your array should contain **10,000 entries.**

hashItems: Create a method that will hash the Strings to an integer value. *You may **not** use any Java hashcode function or any other kinds of libraries.*

Put: Create a method that will take a given String, hash the String, and append a ListNode object containing that String to the end of the Linked List at the index that the hashing function produces.

Readfile: Create a method that will read the dictionary file that was provided to you, hash the Strings and place them into the table using the put method that you wrote above.

ContainsKey: Create a method that will return a boolean value that indicates whether a supplied key exists in the dictionary or not.

Step 3 – The Game (Insert 3 tokens):

Create a class called 'Game.' This class will contain your main method. In this class (if it isn't already obvious) we are going to play a small game. The game involves word permutations. Given two random words from the dictionary of **equal length**, a source and destination word, is it possible to permute the source word into the destination word in **5 hops?** You are only allowed to alter one letter at a time in the source word and the number of hops indicates the number of alterations.

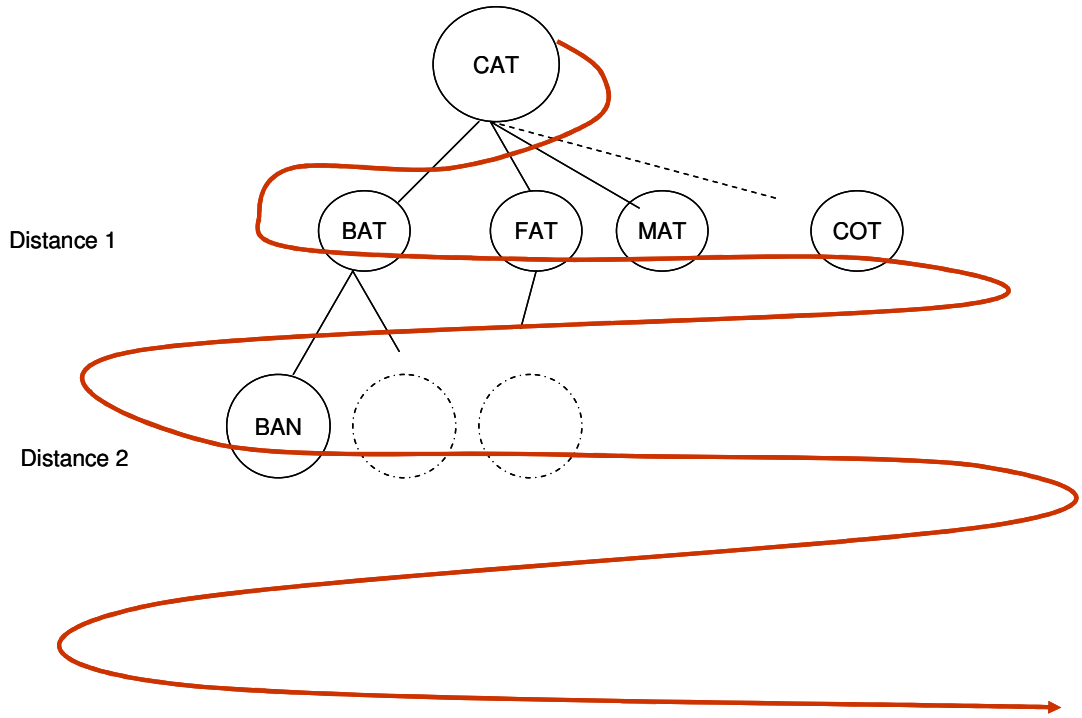
The word 'cat' can be altered into the word 'dog' in 4 hops: cat, cot, dot, dog (there may be other solutions, but this is just one example). In each step only one letter can change.

StartGame: Create a method called 'StartGame.' Using the source word and the destination word, you are going to perform a Breadth First Search, using the source word as the root of your tree. Each level of the tree will contain permutations of the word in the level above it.

The way in which your tree is being created is by generating all the permutations of the source word when you are performing the Breadth First Search. You do not need to actually create a Tree data structure anywhere to store your words. The implementation of your Breadth First Search will take care of this for you.

If 'cat' is at the root, the level directly below will contain all permutations of the word 'cat' that exist in the dictionary (fat, hat, gat, etc.) where you will need to use your Hash Table to determine whether the word exists in the dictionary.

Figure 1: Parent of BAT is CAT. Parent of BAN is BAT etc.



QUEUE: CAT BAT FAT .....COT .... BAN .....

There are many ways of performing a Breadth First Search, but for the purposes of this project, we are going to be using a Queue data-structure. A Breadth First Search works by searching a tree level by level. In the case of this project, you will perform the Breadth First Search as a means of finding your destination word. If you find your destination word, stop performing your search and print the path to get from source to destination word (this is defined in step 4). If the word doesn't exist within 5 hops of your source word, then print a message saying that it cannot be done.

Step 4 - Small Segue:

QueueTree: Create a class of this name. This class will serve as a modified version of a standard Queue data-structure which be used to perform the Breadth First Search. Your QueueTree class will store objects defined by the class below. Create all the standard methods that you feel are necessary (enqueue, dequeue, isEmpty, ifFull, etc). However, you need to make a small change to your dequeue method. For the purposes of this project, you want to keep all the nodes that you have traversed (rather than tossing them out). So, your dequeue method should retain the old head rather than deleting it from the queue (HINT: change what the head pointer points to).

Tnode: Create a class called Tnode. These are the types of objects that will be used in your queue. The Tnode should store an item, the next node in the queue, and should also store another variable parent (this should be of type Tnode as well). Parent is basically to indicate the word that preceded a given permutation. So, in the above case, the permutation of 'cat' will have parent pointing to 'cat.'

The tree in Figure 1 will appear like the image below when you begin inserting the words into the queue:

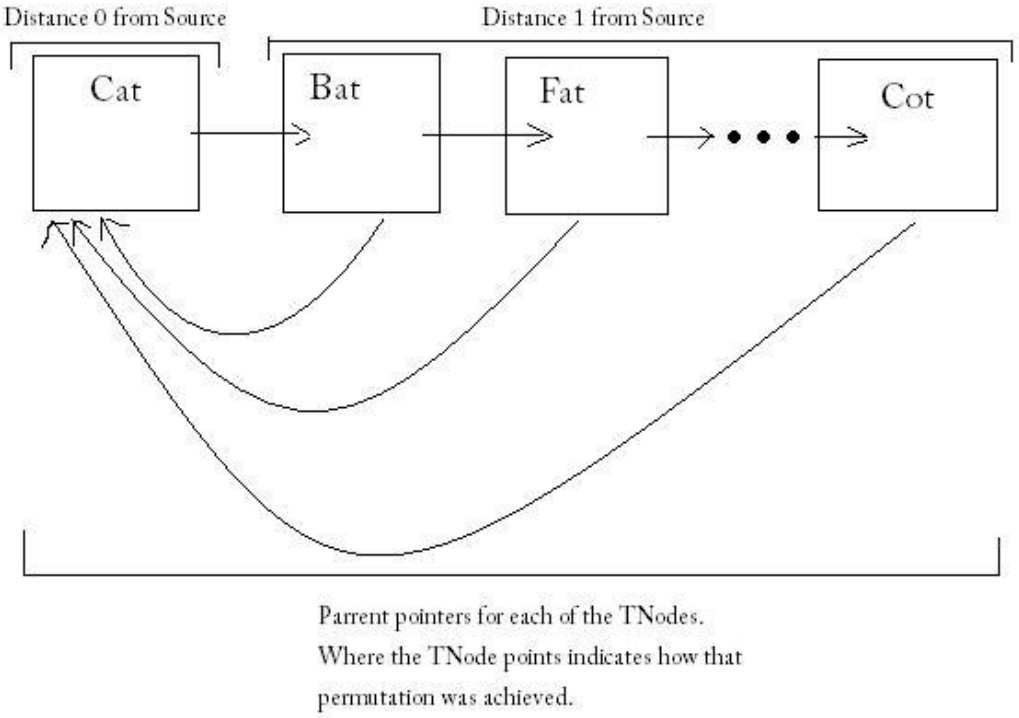


Figure 2: Nodes from the tree in Figure 1 as they appear during the Breadth First Search.

PrintPath: If you do determine that a path exists, **print the path** of how to get from the source to destination word. Write this method within the QueueTree class.

Step 5 – Simulating your Game:

To simulate your program, take user input from System.in in order to test various source-destination pairs.