**Program 3 Fall 2016**

**20 Points**

**Word Ladder Revisited AVL Trees**

PART 1

You have been given the AVL tree code from your author as a starting point. Reading code is a valuable experience. On the job, you are often asked to modify/maintain existing code. You can’t start over and do it your way. You must incorporate your changes into the exisiting code. **You are expected to understand the code that has been given to you.** Make the following changes:

1. Change all variable names to be meaningful names.
2. Make any changes needed to conform to our style guidelines, including better comments.
3. Implement the function deleteMin. Remove the smallest node in the tree and rebalance. This function is to be your own work, not copied from ANY other source. Note that doing a findMin followed by deleting that value will not work, because of duplicates. Plus, you would learn nothing from that approach.

The data to be stored in the AVL tree is to be generic and work for either an Integer or an Object (of the puzzle variety).

Illustrate that the AVL tree is working properly by running the tests shown in the sample code. The starter code includes a Dwarf class so you can see how the code works with an object you create. Do not skip this step! Debugging on a simple example saves a ton of time!

PART 2

**Use the AVL tree as a priority queue to improve your solution to the word Ladder problem (of program 1).** This is the idea. Suppose instead of looking at solutions in the regular order (which we used in program 1), what if you considered partial solutions which look closer? For example, which of the following is the best ladder to pursue if I want a word ladder from “stone” to “money”:

[stone,atone,alone][stone,atone,axone][stone,atone,atony][stone,scone,scene]

[stone,scone,scope][stone,scone,score][stone,shone,phone][stone,shone,shine]

[stone,shone,shore][stone,shone,shote][stone,shone,shove][stone,stane,stade]

[stone,stane,stage][stone,stane,stake][stone,stane,stale][stone,stane,stare]

[stone,stane,state][stone,stane,stave][stone,stane,stand][stone,stane,stang]

[stone,stane,stank][stone,stoae,stoai][stone,stoae,stoas][stone,stoae,stoat]

[stone,stoke,smoke][stone,stoke,spoke][stone,stole,stele][stone,stole,stile]

[stone,stole,style][stone,stope,slope][stone,stope,stipe][stone,stope,stupe]

[stone,stope,stops][stone,stope,stopt][stone,store,snore][stone,store,spore]

[stone,store,swore][stone,store,stere][stone,store,stork][stone,store,storm]

[stone,store,story][stone,stony,stogy][stone,atone,agone,agene][stone,atone,agone,agons]

**Explain why you think one choice is better than others?**

**Note, we don’t know that the one that looks the best IS the best. We are just saying, we will look at our best choice in each round (but keep the other choices in case they start to look better).**

A priority queue is a queue such that insertions are made in any order, but when you remove a node from the queue, it is the one with the best priority. Our priority will be an estimation of the total amount of work needed to solve the problem – so a lower score is preferred. Thus, we will first consider ladders that “look better” to us.

In our previous solution, (program 1) , we considered all one-move solutions before considering all two move solutions, and so on. If we are smarter in which solutions we consider, we can improve the functioning of the solution. In this assignment, you will compare your previous solution to this new technique.

We extend our WordInfo to contain

* priority (moves + estimated cost to reach a solution)

**Best-first search.** We describe a solution to the problem that illustrates a general artificial intelligence methodology known as the [A\* search algorithm](http://en.wikipedia.org/wiki/A*_search_algorithm). For each possible choice, we store the “expected total work” to reach a solution. This expected total work is termed the “priority”. Thus, lower is better.

First, insert the initial state into a priority queue. Then, delete from the priority queue the state with the smallest priority, and insert onto the priority queue all neighboring states (those that can be reached in one more move) using an appropriate estimated cost for each. Repeat this procedure until the state removed from the priority queue is the goal state. The success of this approach hinges on the choice of estimated cost or *priority function*.

**You can compute your “expected work function” any way you want as long it is reasonable and underestimates the real cost.** **Be creative.** Since our expected work underestimates the real work, as soon as we dequeue a ladder which has the goal state, we have a ladder of the shortest possible length. When we dequeue, we know everything else we will enqueue in the future will take at least as many moves to reach the goal.

Here is one choice for computing priority:

* the Length of the ladder (work so far) + number of letters that are NOT correct (minimal number of steps to goal)

Feel free to do something more exotic if you wish.

**Output:**

Since we want to compare this version with our brute force solution in program 1, you will need to have both methods working.

1. Show the output for various inputs using both methods (brute force and A\*). Your output should include
   1. The final word ladder
   2. Length of final word ladder
   3. Which method was used
   4. total number of states enqueued.
   5. Test that the length of the word ladder is the same for both

For example, my output looked like:

Seeking a solution from kiss ->woof Size of List 3862

[kiss kins wins wons woos woof ] total enqueues 2435

Seeking a A\* solution from kiss ->woof Size of List 3862

[kiss wiss wits wots woos woof ] total enqueues 196

1. Find an example for which your “more intelligent” method saves significant time over the brute force method of program 1. Show the same output for this example.

**Show the output for at least the following:**

* kiss woof
* cock numb
* jura such
* stet whey
* rums numb
* An Example of your choosing

**Hints**

HINT 1: Be methodical about debugging. Create a small data set to test with. I called mine “tinyDictionary.txt”. (It can have made up words if that is easier.) Print out every partial solution so you can see if it is doing things correctly.

During debug, when using the A\* search, show each ladder as you pull it off the queue. Be sure to print its priority.

Hint 2: There is one catch. In program 1, we wouldn’t reuse a word in a word ladder if it was already in a word ladder. The thinking was this. We processed word ladders in order of length. If we previously had a word ladder of length 4 ending in “clone”, there was no point is generating a longer ladder ending in “clone” because it would certainly be worse.

HOWEVER, if we don’t always work on the shortest ladder (but work on the “best” solution), we may lose a good solution by not allowing word reuse.

This was my solution. I kept track of the position in the word ladder where a word was used. If I ever wanted to use it at an EARLIER position, I allowed it. In most cases, it didn’t matter. BUT, in some cases it did.