**Program 3 Fall 2016**

**20 Points**

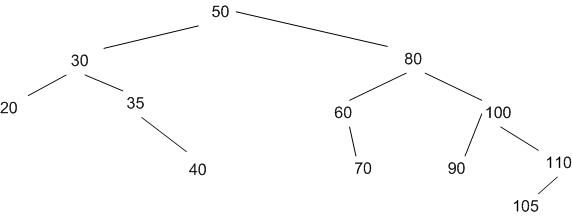
**Word Ladder Revisited AVL Trees**

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.
3. Write a toString function which creates an indented version of the tree (similar to that of program 2, but for a binary tree).

For example, if your AVL tree looks like:



I would print it as

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | 110 |  |
|  |  |  |  | 105 |
|  |  | 100 |  |  |
|  |  |  | 90 |  |
|  | 80 |  |  |  |
|  |  |  | 70 |  |
|  |  | 60 |  |  |
| 50 |  |  |  |  |
|  |  |  | 40 |  |
|  |  | 35 |  |  |
|  | 30 |  |  |  |
|  |  | 20 |  |  |

The idea is that I’ve printed the tree on its side. Turn the paper to the side and draw the arcs and the two versions match.

1. Create a toString2 function which prints the tree as a list of elements.
2. Implement the removeMin function. Remove the smallest node in the tree and rebalance. This function is to be your own work, not copied from ANY other source.

The data to be stored in the AVL tree is to be generic and work for either an integer or a WordLadder.

Illustrate that the AVL tree is working properly by printing the tree (using toString) after each of the following groups of operations:

* Add: 1 3 5 7 9 9 9 11 2 9 4 8 (now print tree)
* Remove 7 9 (now print tree)
* Add 30 50 30 30 15 18 (now print tree)
* Remove Min (now print tree)
* Remove Min (now print tree)
* Remove Min (now print tree)
* Add 17(now print tree)

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 potential solutions in the regular order (which we used in program 1), what if you first 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 define a *WordLadderState* to contain at least:

* wordLadder
* priority (cost so far + 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 dequeue 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 ladders dequeued
   5. total number of ladders enqueued.

For example, my output looked like:

[cock,rock,rocs,roms,rums,rump,dump,dumb,numb] 9 ASTAR dequeued 1810 enqueued 2995

[cock,dock,duck,duce,dure,dura,duma,dumb,numb] 9 BRUTE dequeued 3627 enqueued 3715

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
* 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: Print output both to the console and to a file. Then, you can easily remove the printing to the console (to make it run faster and easier to read).

Hint 3: 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.

HINT 4: Getting the same code to work for ints and WordLadderStates forces us to be more methodical in our approach. For example:

The toString2 function looks something like:

string toString2( AvlNode \*t) const  
{  
 stringstream ss;   
 if( t != nullptr )   
 {  
 ss << toString2(t->left);  
 ss <<t->element << endl;  
 ss << toString2(t->right);  
 }  
 return ss.str();  
}

This works fine if element is an int, but not so great if element is a WordLadderState.. How do you get this to work? Ints don’t have an element component. WordLadderStates don’t know how to print themselves.

AHHHHH.  You need to overload << for WordLadder

It is done like:

ostream& operator<<(ostream& ss, const WordLadder & gs) {

ss << gs.toString() << endl;

return ss;

}

Note however, that this procedure is not (and cannot be) a member function of WordLadder because the first parameter of << needs to be an ostream.

**What to Turn in:**

Submit a zip file containing your software project. If you are using visual studio, submit the entire project (so graders can just click on the project name to run it.) The zip file should contain a readme file which tells how to run the program (what IDE you used).