**SpellCheckTree**

// The 314,262 words in dictionary.txt are sorted alphabetically.  
// - Original source is a Linux spell-check file, minus profanity.  
// - There are notable omissions in the list (e.g., aardvark).  
// - Feel free to add your own words.  
//  
// Checking if two strings are equal is O(K) for a K-letter string.  
// - Assuming “equals” does a simple loop over the letters.  
// - There are tricks to do this in constant time.  
//  
// Searching the flat list for a string is O(K\*N) if there are N words.  
// - Assuming “contains” does a simple loop over the words.

//  
// Binary search could do this in O(K\*log(N)) time.  
//  
// A tree of letters searches in O(K)—also, takes less memory storage.

// - not a binary tree!

// - need an end-of-word marker.

// - For N>1000 words O(K) is at least an order of magnitude better.

SpellCheckTree uses the SpellNode class.

SpellNode stores a char value and an ArrayList of SpellNodes. Processing each SpellNode needs more than “left/right.” Therefore, a tree built of these nodes is NOT a binary tree. Here is a picture of the first three words in the list,

aback  
abacus  
abaft

class SpellNode  
{  
 char val;  
 ArrayList<SpellNode> chldrn;   
 . . .

}

If the textfile consisted of only those three words, the run prints:

Total Letters: 16  
 Total Nodes: 13  
 Savings: 19%

Search for word:

10 class SpellNode   
 11 {  
 12 public char val; //Mr. Torbert likes public fields  
 13 ArrayList<SpellNode> chldrn;   
 14   
 15 public SpellNode(char ch)  
 16 {  
 19 }  
 22 // Searches for ch in the children of this node.  
 23 // if pos==chldrn.size() then ch was NOT found   
 24 int pos\_of\_char(char ch)  
 25 {  
 33 }

34 public String toString()  
 35 {  
 37 }  
 38 }  
   
 44 public class SpellCheckTree  
 45 {  
 46 public static void main(String[] args) throws FileNotFoundException  
 47 {  
 48 int list\_total=0, tree\_total=1;  
 49 SpellNode root=new SpellNode('\*'); // dummy value for root  
 // Big loop:

// read a word

// count the total number of letters

// for each letter in the word  
 // get the position of the letter in the array

// if the letter isn't there

// add a node and count.   
 // get the next level.  
 // add the node for end-of-word marker, count.  
 // Print the statistics: total letters, total nodes, %   
 // Big loop:

// prompt for a word.

// set is\_a\_word to true.  
 // walk down the tree going from letter to letter.

// get the position of the letter in the array

// if letter is not in the array, set the boolean, break.

// get the next level

// if we stayed in the tree, make sure we end up with '\*'.  
 // print "Yes, it's a word."  
 // else  
 // print "Sorry, that is not a word."

// Recursive part: Double-check the tree\_total value after the tree has

// been built; requires a "simple" walk of a non-binary tree.