About Lab 9

In Lab 9 you will implement a Boggle game. The lab provides the graphical interface; you need to build the game engine that responds to guesses from the user.

In Boggle the player (we will make only a 1-player game) rolls a set of 16 dice containing letters on each face. These are put into a 4x4 grid. The player makes words out of this grid by starting with one of its letters, then adding letters that are adjacent in the grid.

For example, with grid

A	S	D	M
P	T	E	B
R	O	F	S
I	G	T	N

we can find words

past:

A	S	D	M
P	T	E	B
R	O	F	S
I	G	T	N

got:

A	S	D	M
P	**T**	E	B
R	**O**	F	S
I	**G**	T	N

trot:

A	S	D	M
P	**T**	E	B
R	**O**	F	S
I	G	**T**	N

spot:

A	**S**	D	M
P	**T**	E	B
R	**O**	F	S
I	G	T	N

fog:

A	S	D	M
P	T	E	B
R	**O**	**F**	S
I	**G**	T	N

and plenty of others

There are two different aspects to your work for this lab. One part is a data structures issue. We need to be able to tell whether a word is in a list -- a dictionary list of words, a list of words the user has already found, a list of words that actually exist in the grid, and so forth. We will use a trie for all of these questions.

Remember that each node of a trie represents a prefix of the words in the tree. Each node has 26 children, which we think of as indexed 'a' to 'z'. To either find or insert a word into the structure, we walk along the letters of the word; if the next letter is 'p' we go to the child of the current node whose index is p. At the end we have a boolean flag that says whether or not the string we have walked along is a word contained in our structures.

So you need to implement class myTrie. Most of this is very straightforward. One issue you will need to face is how to convert a single character into an index in the array. You want character 'a' to be mapped to 0, 'b' to 1 and so forth. One way to do this for character ch is to look at ch-'a'. An alternative way is to define

```
        static String alphabet =
"abcdefghijklmnopqrstuvwxyz";
```

and look at alphabet.indexOf(ch).

In either case you need to beware of characters that don't give an index in the range 0..25. Capital letters, hyphens, apostrophes and other characters can do this. You don't want your program to crash, regardless of what input it finds.

Don't be confused by the structure of the class. myTrie is actually the node class for a trie. There is a class variable size that holds the number of words stored at this node or below. The add method is recursive and returns true if the add resulted in a new word being added to the trie; when you come back from a recursive call you can adjust your node's size variable. An alternative, though less efficient, method for implementing the size() method for the trie is to ignore the class size variable, and instead return the size of the list representation of the trie.

The myTrie class has a private method that returns a list containing all of the words stored in the trie. This is very useful. A simple recursion builds the list. There is also a method that returns an iterator for the trie; just make a list representation and return its iterator.

You should find the myTrie class fairly simple to implement. The other, and more challenging, part of the lab is the implementation of the Boggle class, which is the game engine.

The Boggle class maintains 3 tries:

- lex, which contains all of the words in a dictionary you read from a file at the start of the program
- foundWords, which contains all of the words in the dictionary that can be made from the current values on the 16 letter dice. After the dice are rolled you walk around the board looking for the words to put here.
- guesses, which contains all of the words the user has found so far

The constructor for the Boggle class does a lot of things:

a) It reads a dictionary file into the trie lex.
b) It calls a method fillDice( ) that reads a file that contains the letters that are on the 16 dice (the dice aren't all the same).
c) It calls a method fillBoardFromDice(). This shuffles the 16 dice, then randomly chooses one of the 6 sides of each die to appear on the board.
d) Finally, it calls a method fillFoundWords( ) that finds all of the words in the dictionary that can be built from the current board, and places them in the foundWords trie. More about this later.

The lab document gives you the main() method to use in the Boggle class. This method hooks your data structures up to a graphical front-end that lets a user play the game. The front-end assumes you have all of the methods of the Boggle class implemented, so you won't be able to run it until you have at least a stub for each of these methods. In fact, it is hard to do any effective debugging of the Boggle class until you have it completely written.

Here is an easy way to shuffle an array A. This is called the "Fisher-Yates Shuffle":

```
rand = new Random();
n = A.length-1;
while (n > 0) {
        m = rand.nextInt(n);
        // switch A[m] and A[n]
        n = n - 1;
}
```

There are two algorithmic methods that make up most of your work on this part of the lab"

- MyTrie search(Square sq, String prefix) This returns a trie with all of the words on the board that are in the lex trie and can be completed from the prefix, starting on square sq.   The fillFoundWords( ) method that creates the foundWords trie calls this 16 times (once for each square on the board) with the empty string as prefix.

- ArrayList<Square> squaresForWord(String w)

  This is called when the user claims that w is a word on the board. You need to find a sequence of squares that represent legal moves and contain the letters of w. This is used to highlight the squares on the board.

The lab document makes a suggestion for the search method, which we show on the next slide

The "marking" prevents you from re-using letters you have already used.

MyTrie search(Square sq, String prefix)

```
// check to see if we have found a word on the current path
if the current path represents a word in the dictionary
    add the word to the wordlist
// continue searching on all possible paths from this square
if there are any words possible from this prefix
                  (use lex.containsPrefix())
    for each unmarked square s adjacent to sq
        mark s
        recursively search for words starting at square s
                  using prefix prefix+sq.letter,
       add these to wordlist
       unmark s
```

You are more on your own for the squaresForWord( ) method. Here is some help:

ArrayList<Square> squaresForWord(String w) calls a helper method:

ArrayList<Square> squaresForWord(Square sq, String w)

The top level squaresForWord(w) calls the helper method 16 times, once for each square of the board. The first thing the helper method does is to check whether Square sq contains the first letter of w; if not, it returns an empty list. If it does, it recurses on each of its neighbors, using string w.substring(1). You want to use a marking scheme, just like the search( ) method, to prevent reusing squares. If the recursion comes back with a list of the same length as the substring, you know you have found the list of squares; just add the current sq to the start of it and return it.

There are two things about this that you might find tricky. One is the indexing scheme for the board. The Square class has class variables x and y for the location of the square on the board. If you think of an array in the usual notation, board[row][col] is one entry of the board. The square class has x=row and y=col, so Square s is s = board[s.x][s.y]. Don't think of x as the horizontal variable and y as the vertical one, but rather think of the squares as board[x][y].

If you get this twisted, you will see the wrong squares on the board highlighted when you correctly find a word.

The other tricky thing is finding the neighbors of a given square sq. There are more formal ways to handle this, but here is some easy code:

```
for (int dx = -1; dx <= 1; dx++) {
    for (int dy =-1; dy <= 1; dy++) {
        if (dx == 0 && dy == 0)
           continue;
         int newx = sq.x + dx;
         int newy =sq.y + dy;
         if (newx >= 0 && newx < 4 && newy >= 0 && newy < 4) {
            ....
         }
    }
 }
```

There are two error messages you can get from the game. When you guess a word, the game first calls squaresForWord( ) to see if it can find squares for your word. If your method returns an empty list of squares, the game says "**word not on board."** If you do return an appropriate list of squares, the game looks for the word in your trie foundWords. If it is not there, it replies "**word not in lexicon**". Note that it does not look in the dictionary trie lex for the word. If your search method is incorrect, which means your foundWords trie will be incorrect, you might get this message for correct words that are actually on the board.