Lab Goals

- Implement a de la Briandais Trie
- Use your DLB Trie to find words in a Boggle board

Assignment Details

Before break, we discussed string searching, and discovered that the most efficient way to search for whether or not a string exists is to use a trie. The default trie implementation is a set of nodes, such that every node holds a value and an array of references to every possible next character in the alphabet. Because many of these references will be null deep in the trie, an alternative is to use a de la Briandais trie, in which the node consists of a key/value pair, a reference to a “child node” which is the next letter in the string, and a reference to a “sibling node” which is an alternate character at the current location in the string.

In this lab, we will implement a de la Briandais trie, and use it to search for all words in a Boggle board. Note that because of Exam 2 (and partly due to the length of this lab), this will be a two-week lab.

Part One: Creating the de la Briandais Trie (20 points)

The first component of this lab is to implement your de la Briandais trie. First, you will need to create some variant of a dlbNode class, which will store the key/value pair and the child and sibling references to other nodes. Then, a full DLB class can be implemented to use these dlbNodes to store characters.

In the lab8/data directory is a text file called dictionary.txt, which contains all of the words that we will store in our DLB. Because we are not concerned with storing a value for each of our words, you can design the DLB so that a node that corresponds to the end of a word will have a value of 1, while a node that is internal to a word will have a value of 0. This will allow you to efficiently check whether or not a string is in fact a valid word.

You should open this dictionary.txt file and scan it in line-by-line, adding nodes to the DLB and updating node values to 1 as appropriate. When complete, your DLB will contain the English language in a very compact form!
Your DLB should (at the very least) contain a function `add(String input)` to insert a new word into the DLB, as well as a function `search(String input)` to determine whether or not a word is stored in the DLB. In the `lab8/src` folder is a program called `DictionaryTest.java`, which will load the `dictionary.txt` words into your DLB, then search through the DLB for a set of strings, printing “TRUE” if the word is found or “FALSE” if it is not. You may need to modify this program slightly to make it work with your DLB implementation, but it is provided for you to prove to me and to yourself that your DLB is functioning properly.

Part Two: Using the DLB for Boggle (20 points)

Similar to the Eight Queens lab, you will now implement a recursive, backtracking algorithm to find all of the words stored in an \( n \times n \) Boggle board. If you have never played Boggle, the rules are simple. A game board is generated or loaded, consisting of a single letter stored inside of each cell in an \( n \times n \) array. We can search for words by tracing paths left, right, up, and down between the array cells. For example, consider the game board in Figure 1.

```
F R O O
Y I E S
L D N T
A E R E
```

Figure 1: A sample Boggle board

Beginning in the top-left corner, we can identify the word “FRIEND” by following links right, down, right, down, and left. We can extend “FRIEND” to “FRIENDLY” by adding two more links, first to the left, then up. We cannot include the word “FROST” because diagonal steps are not allowed. Nor can we include the word “FRIENDS,” because although the letters are all connected, there is no direct path between the D at \([2][1]\) and the S at \([1][3]\). We also also not allowed to wrap around the edges, so “STEAL” is not allowed (beginning at position \([1][3]\)).

To implement a recursive, backtracking algorithm, we need to consider each array cell as a possible starting location. From there, we can try to add a letter going up, going right, going down, and finally going left (or really in any other order). If these are recursive calls, then each recursive call will have an independent copy of the string to build off of. We would want to stop the recursive calls when a null is reached in the DLB. For example, starting at the F position of Figure 1, we cannot go up, so we will make a recursive call to FR. FR is not a word, but the string does exist in the DLB, so we can continue. We cannot go up from the R, so we will make a recursive call to FRO. FRO is also not a word, but the string does exist in the DLB, so we can continue. We cannot go up from the O, so we will make a recursive call to FROO. Unfortunately, FROO does not exist in the DLB, so we will backtrack to the FRO level, and try to go down, making a recursive call to FROE. This is also not found in the DLB, so we will backtrack to the FRO level, and try to go left.
Here is where our rules differ from standard Boggle. In standard Boggle you cannot repeat a letter. However, keeping track of letters in the current string being built can get a little tricky, so we will allow it. This means that in the Figure 1 board, the word “DID” is legal (starting at [2][1]). It also means that we can make a recursive call to FROR. FROR also does not exist in the DLB, so we need to backtrack to the FRO level. Because we have tried all of the possible directions for FRO, we backtrack again to the FR level, then try to go down to the FRI. FRI is now a word, but does exist in the DLB, so we can continue. This pattern will continue throughout the full board.

The goal for this component of the lab is to print out all words that are stored in the board. Alternatively, you could save all of the words from the board into an ArrayList (or a similar data structure), then let a user play Boggle, checking the words that they enter against this ArrayList of solution words. Either option is possible, and is up to you.

In the lab8/data directory are three board files (data1.txt, data2.txt, and data3.txt) which contain board files stored as a single string. The first two files are 16 characters in length, representing a 4 x 4 board, and the third is 25 characters in length, representing a 5 x 5 board. You can read each of these files character-by-character to build sample boards. Also in the labs8/data directory is a file out1.txt, which lists all 188 words that you should find in data1, to prove to yourself that your DLB is working with your Boggle game, and that your recursive, backtracking algorithm is not missing anything.

Part Three: While You Have Some Downtime (10 points)

Answer the following questions thoroughly:

1. We can calculate the compression ratio of a compression scheme by using the simple formula 
   \[ \frac{\text{bits with compression}}{\text{bits without compression}} \]. What is the compression ratio of the example shown on Slide #13 of Lecture #18? Try it with both 7 and 8 bits per character for the uncompressed text.

2. Prove that the two longest codewords in a Huffman code have the same length.

3. Create a Huffman code for the string “AAAAAAAAAABBBBCDDDDDEFFG!” What is its compression ratio, with both 7 and 8 bits per character uncompressed?
Submission Details

For this lab, please submit a paper copy of everything listed below. Additionally, please upload all of your files to a folder in your BitBucket repository clearly labeled as “Lab 8.”

1. Your source code for the DLB in Part One and the Boggle Solver in Part Two
2. The output to the DictionaryTest.java file, to show that your DLB works (Part One).
3. Sample output from either data2.txt or data3.txt, showing that your program can find all words stored in a Boggle board (Part Two).
4. Your answers to the questions in Part Three.
5. An Assignment Information Sheet filled out for your source files.

Please remember that all files that you submit should be your own work, though you are welcome to discuss high-level topics and algorithms with classmates.