

Assignment 3 – Boggle Solver

Credit Todd Feldman for the original idea behind the assignment. Adapted from handouts of Julie Zelenski and Eric Roberts.

Boggle Solver

The Boggle board is a 4x4 grid onto which you shake and randomly distribute 16 dice. These 6-sided dice have letters rather than numbers, creating a grid of letters from which you can form words. In the original version, the players start simultaneously and write down all the words they can find by tracing by a path through adjoining letters. Two letters adjoin if they are next to each other horizontally, vertically, or diagonally. There are up to eight letters adjoining a cube. A grid position can only be used once in the word. When time is called, duplicates are removed from the players' lists and the players receive points for their remaining words based on the word lengths. In this assignment, you will be creating a program that will find all the words on a boggle board.

This assignment is broken into two parts. The first part of the program will be creating a dictionary that can be used to store and look up words. This dictionary implementation will use a special tree called a prefix tree.

```
C:\Users\Erickson\Documents\Visual Studio 2015\Projects\S
127142 words loaded.

Enter Board
-----
Row 0: a u c o
Row 1: n l n i
Row 2: o s a e
Row 3: m a i e

Show Board (y/n)?: y
Word: ala
Number of Words: 1
'a' u c o 1 0 0 0
n 'l' n i 0 2 0 0
o s 'a' e 0 0 3 0
m a i e 0 0 0 0

Word: alan
Number of Words: 2
'a' u c o 1 0 0 0
n 'l' 'n' i 0 2 4 0
o s 'a' e 0 0 3 0
m a i e 0 0 0 0

Word: alane
Number of Words: 3
'a' u c o 1 0 0 0
n 'l' 'n' i 0 2 4 0
o s 'a' 'e' 0 0 3 5
m a i e 0 0 0 0
```

```
C:\Users\Erickson\Documents\Visua
127142 words loaded.

Enter Board
-----
Row 0: a u c o
Row 1: n l n i
Row 2: o s a e
Row 3: m a i e

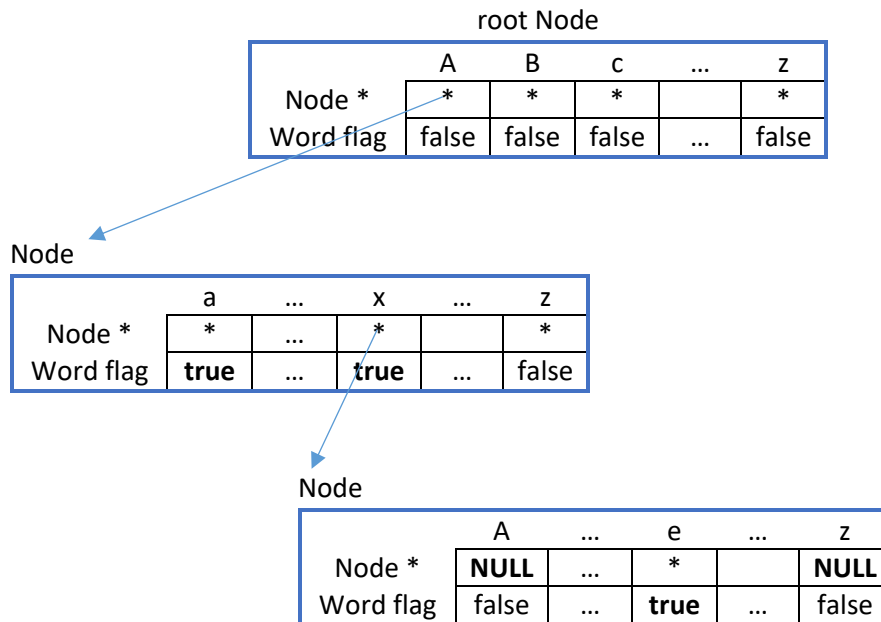
Show Board (y/n)?: n
1 ala
2 alan
3 alane
4 alans
5 alae
6 alas
7 als
8 also
9 anlas
10 ansa
11 ansae
12 anoa
13 anoas
14 unco
15 unci
16 uncia
17 unciae
18 uncial
19 uncials
20 unai
```

Part 3a: Dictionary

Prefix Data Structure

To store the words in this assignment, you will be creating dictionary using a data structure called a prefix tree. This data structure allows for the quick lookup of whether or not a word is valid. It will also allow you to find all words with a specific prefix.

Rather than store each word individually, we instead store the words using a tree:



The example above shows two how the words “aa” (abbreviation for administrative assistant) and “axe” are represented using a tree. Each node holds 26 pointers to other nodes; each of these nodes corresponds to a specific letter. Each node also holds an array of 26 boolean flags.

Essentially, each word is represented by a path down the tree. If the path ends with a “false”, then the path does not represent a valid word. For example, the root node has a path from the first Node * pointer (i.e. the pointer representing ‘a’) to another Node. Notice that this second level node has a “true” flag for the first index. This indicates that “aa” is a valid word. If we examine the pointer for the second level ‘x’ position, we find that a path exists for “ax” to another node. When we examine this third level node, we find that the pointer for the ‘a’ position is NULL. This indicates that “aaa” is not a valid word. If we examine the pointer for the ‘e’ position, we find that the flag for this position is true; this make sense since “axe” is a valid word.

Prefixes

A prefix is a valid path that may or may not be a valid word. For example, “a” and “ax” are valid prefixes since there are words starting with these letters. However, “aaa” is not a valid prefix.

Milestone 1 – Implement and test the Dictionary

Your first job is to implement the Dictionary class. The following is the header file:

```
struct Node {
    // Your data structure goes here.
};

class Dictionary
{
public:
    Dictionary();
    Dictionary(string file);
    void addWord(string word);
    bool isWord(string word);
    bool isPrefix(string word);
    void PrintWords(string prefix);
    int wordCount();

private:
    Node * root;
    int numWords;
    // Your private helper methods go here
};
```

Constructor

There are two versions of the constructor. The first just establishes an empty node. The second constructor takes in a string file and reads the file line by line and uses the addWord method to add words to the dictionary.

addWord

This method adds a word to the tree. This is accomplished by reading the word character by character and creating a path for the word if it doesn't exist. Below is the pseudocode for this method:

```
currNode = root;
```

```
for (each character c of the word) {
```

```
    if (the branch for character c is NULL) {
        set the branch of character c = new Node.
        set all the flags of this new Node to false
    }
    currNode = the pointer index of character c
```

```
}
```

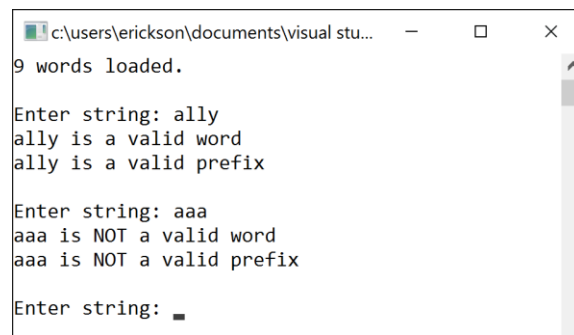
Set the flag at the currNode for index of character c to true

isPrefix

This method returns true if a path down the branches exists for a word. This method will have an implementation very similar to the addWord method. It should return false if it runs into a NULL. If it manages to get through the entire loop without hitting a NULL, it should return true.

isWord

This method is very similar to the addWord and isPrefix method. If you have already implemented the isPrefix method, then it should be fairly trivial to implement isWord. The main difference is you should return the flag found at the end of the path.



```
c:\users\erickson\documents\visual stu...
9 words loaded.

Enter string: ally
ally is a valid word
ally is a valid prefix

Enter string: aaa
aaa is NOT a valid word
aaa is NOT a valid prefix

Enter string: 
```

Random Hints for the Dictionary Class

Be sure to test using the small dictionary before using the bigger dictionary. Here are some random hints that you may find useful.

Reading a file line by line

The following code reads a file one line at a time and prints out the line:

```
ifstream myFile(file);

numWords = 0;
string line;
while (getline(myFile, line)) {
    cout << line << endl;
}
```

Figuring out the index of a character

It will be useful to be able to index each character:

```
0 1 2 3 4 5 6 ... 25
a b c d e f g ... z
```

We can take advantage of the fact that characters can easily be cast into integers and use the following to figure out the index of a particular letter. For example, the following could be used to figure out the index of the character e:

```
(int)'e' - (int)'a'
```

The above would evaluate to the integer 4.

Part 3b: Boggle Solver

Once you have completed and tested your Dictionary class, your next task is creating the Boggle solver.

User Input

After creating and loading the dictionary, you should prompt the user for input:

```
C:\Users\Erickson\Documents\Visual Studio 2015\Projects\S
127142 words loaded.

Enter Board
-----
Row 0: a u c o
Row 1: n l n i
Row 2: o s a e
Row 3: m a i e
```

After initializing the board, you should ask the user whether or not to print the board when finding the solutions:

```
C:\Users\Erickson\Documents\Visual Studio 2015\Projects\S
127142 words loaded.

Enter Board
-----
Row 0: a u c o
Row 1: n l n i
Row 2: o s a e
Row 3: m a i e

Show Board (y/n)?: y
Word: ala
Number of Words: 1
'a' u c o 1 0 0 0
n 'l' n i 0 2 0 0
o s 'a' e 0 0 3 0
m a i e 0 0 0 0

Word: alan
Number of Words: 2
'a' u c o 1 0 0 0
n 'l' 'n' i 0 2 4 0
o s 'a' e 0 0 3 0
m a i e 0 0 0 0

Word: alane
Number of Words: 3
'a' u c o 1 0 0 0
n 'l' 'n' i 0 2 4 0
o s 'a' 'e' 0 0 3 5
m a i e 0 0 0 0
```

```
C:\Users\Erickson\Documents\Visua
127142 words loaded.

Enter Board
-----
Row 0: a u c o
Row 1: n l n i
Row 2: o s a e
Row 3: m a i e


Show Board (y/n)?: n
1 ala
2 alan
3 alane
4 alans
5 alae
6 alas
7 als
8 also
9 anlas
10 ansa
11 ansae
12 anoa
13 anoas
14 unco
15 unci
16 uncia
17 unciae
18 uncial
19 uncials
20 unai
```

Board Printing

One of the requirements of the program is that the solver should also show the path of the solution. For the example below, we can see how “alane” is found using the grid on the right.

Word: alane							
Number of Words: 3							
'a'	u	c	o	1	0	0	0
n	'l'	'n'	i	0	2	4	0
o	s	'a'	'e'	0	0	3	5
m	a	i	e	0	0	0	0

step



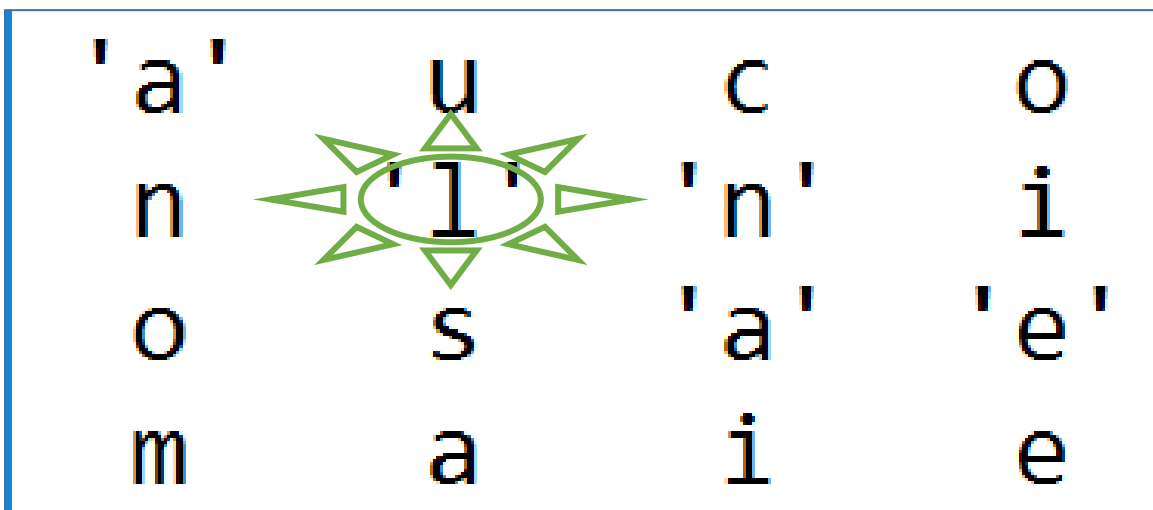
Recursion Strategy

A straightforward strategy to finding all the words on the board is to use recursive backtracking. The general idea is to start at each position and then explore all paths originating at that position being careful not to re-visit grid positions.

The solution will start with the SolveBoard function. SolveBoard will serve as a “wrapper” for another SearchForWord function that will do the heavy lifting of the recursion:

- For each position on the board
 - Call SearchForWord starting at the current board position

The job of SearchForWord is to **recursively** (hint hint...) check the surrounding 8 positions.



I suggest thinking about the base case(s) first. A solution using this strategy is very short (my SearchForWord is only about 40 lines) so if you find yourself writing lot of code, you may want to rethink your strategy. Be sure to make use of the isPrefix method to prevent exploration of paths that will not produce words.

Each call to SearchForWord function will require multiple variables in order to complete its task:

- What row and col to start
- The current step (see diagram for Board Printing on previous page)
- What the current step is (ie the numbers in the grid above)
- A string that stores the current progress of the path
- The board
- A dictionary to determine whether or not you found a word
- A way to remember all the previously visited spaces
- A way to remember words you have already found (Another dictionary would work well)

I strongly suggest you use a debugger or use a generous number of cout statements to help you debug this method.

Extra Credit

- 3% Create an option for the user to roll each of the 16 Boggle dice to create a random board to solve.
- 10% Create an option to play Boggle as a game (minus the egg timer).

Part 3a: Deliverables

- Dictionary

Part 3b: Deliverables

- BoggleSolver and any other files used for the game
- Your test plan for BoggleSolver
- The output file from the test runs