CS917: Foundations of Computing
Lab 2

1 Introduction

In this session you'll be exploring the use of search algorithms, and looking at how the selection of data structures and algorithms can impact the performance.

As always, any necessary resources are available to you on the website.

Have fun!

2 Binary Trees and Traversal

To familiarise ourselves with the tree class, let's create a means of updating and traversing our tree.

On the website you should find an example basic version of a tree, missing an insert function. Create an insert method that adds a value to the correct positions within the binary tree (i.e. if it's less than the current node value place it somewhere to the left, if it's greater than, to the right. We'll assume each value is unique for now). You are welcome to modify the structure/methods and add methods as you see fit. Hint: This suggests we have to create a node for the new value at a leaf position, and update the parent to store it's relationship to the new leaf node.

Once you can insert values successfully, let's verify our tree structure by writing some traversal methods. Create methods to perform the following:

- BFS
- DFS: In-Order Traversal
- DFS: Pre-Order Traversal
- DFS: Post-Order Traversal

Experiment by adding more numbers of your own and verifying your traversal methods work by getting them to print out the values of each type of traversal.

3 Word Tree

We know that we can store words in a list format. However, let's try a different approach. How could we store words in a tree format? (There are a couple of different approaches!) Construct a word store that is in a tree format from the file of words stored on your systems at "/usr/share/dict/words". Here are a few hints to get you started:

- How do I distinguish between the words 'car' and 'far'? In what manner could this be represented in a tree? (Do we store whole words or do we store characters?).
- What does each level in a tree represent? (E.g., I could make it such that following a path in the tree gives me a word).
- Your solution might find that the existing tree skeleton is insufficient for your purposes, since it is only a binary tree. If so, what modifications might you want to make to store more children per node?
- You will need a method to lookup whether a word you are given exists in the dictionary (i.e. a search).

How does this compare to the use of a list for both the speed of the operations and the space used?

4 Anagrams

For a bonus exercise, let's make an anagram generator! Given a word, I should be able to deduce any other words that exist with a permutation of the characters used for the first word.

There are a couple of different ways of doing this, so think about the performance costs and memory costs. Here are a few hints!

- What are the properties of words that are anagrams of another word (think types and number of characters)
- How can we use these properties efficiently? If we were to use the tree from the previous exercise, what would be the cost of finding all anagrams? Hint: Is there a better way that we could ***map*** from the properties of a word to all words with the same properties?