# AE1PGA Lab 10

## Word-guessing game from dictionary

In Lab 6, one of the tasks was to create a word guessing game (Hangman). When the program was run, the first stage was for the user to type in all the possible words to randomly pick from. This is rather tedious. Normally, it can be any possible word in the dictionary. Luckily for us, Linux distributions normally have a dictionary text file which contains all the words in the dictionary, 1 word per line. This is stored at "`/usr/share/dict/words`". Be aware that there might be some non-English characters in the dictionary; if you treat each character as just an ASCII code between 0-255 it will be fine. I suggest filtering them in your program to only use the representing letters 'a' - 'z'.

Create a copy of your Lab 6 program and then modify it so that it reads all the words from the dictionary file into the program, then randomly chooses from them each time a new game starts.

---

---------- Advanced exercises ----------

## Graphs

Create both an adjacency-matrix and adjacency-list implementation of a directed, unweighted graph data-structures. Each data-structure should support operations similar to the ones defined for other data-structures: create a graph, destroy a graph, add vertex/edge, remove vertex/edge, does vertex exist, does edge between two vertices exist, get all vertices, get all vetices connected to a vertex, is there is path between two vertices, etc.

Extension (2-4 hours including research): The `graphviz` package has a various commands (`dot`, `neato`, `twopi`, etc) which take a text file describing a graph and generate an image (jpg, png, svg, etc) of that graph. Add a new function to each graph implementation which takes a filename and saves a graphviz-formatted text file which describes the graph. You can find an overview of the file format by reading the man page (`man dot`) or by searching online. You only need a small subset of the language to be able to save the graph.

## Binary Search Tree

Implement a binary-search tree (BST) data-type. Remember that the difference between a binary tree and a binary search tree is that in the latter, the nodes are stored in a sorted order. For this exercise, you can make it a BST of `int` stored in ascending numerical order. The data-structure should support the following operations:

- `BST *bst_create()` that takes no arguments and returns a pointer to a BST type. You should define the `BST` data-type however you want.
- `int bst_add(BST *root, int val)` that takes an existing BST created by `bst_create` and a new value, and adds that value to the tree. It should return true/false on success/error.
- `int bst_remove(BST *root, int val)` that takes an existing BST created by `bst_create` and a new value, and removes that value to the tree if it exists. If it is not in the tree then the function should do nothing. It should return true/false on success/error.
- `bst_contains(BST *root, int val)` that takes an existing BST created by `bst_create` and a value, and returns true/false if the value is/is not in the tree.
- `void bst_destroy(BST *root)` that takes an existing BST created by `bst_create` and frees any memory associated with it. The pointer to the BST is invalid and should not be used after this function call.

---

*End*