# Spencer CW Au

**Data Structure for synsets.txt:**

I used 2 HashMaps to store the values of synsets.txt, with Map1 having its key corresponding to the integer value of each word, and the value being an ArrayList of Strings to check for collisions, separated by a white space, assigned to it and Map2 having its key corresponding to each unique word String (including the multiple words separated by white space for a single unique int), and the value being an ArrayList of ints to check for collisions. The HashMap was an easy choice as it creates a pair for every unique integer/String entry and the ArrayList value allows it to account for multiple words with the same definition/multiple ints for the same word, and the lookup time given an integer/String is O(1).

**Data Structure for hypernyms.txt:**

I used the digraph to store the values for hypernyms.txt, as every line on the file contains an integer representation for a synset word with subsequent int being the int representation for its respective hypernyms. The digraph was the obvious choice as the project assignment requires us to implement this data structure, the hypernms.txt file contains only int values (in which our implementation of Digraph.java and SAP.java uses a digraph of int values), and that the required outputs (asking for the shortest ancestral path and the lowest common ancestor) were already implemented in SAP.java, which as stated above, utilizes and expects a digraph of ints.

**Algorithm to Compute SAP:**

The algorithm I used to find the SAP (shortest ancestral path) found the intersection between two vertices given the path between that respective vertex and the root vertex. Utilizing the methods provided by the BreadthFirstDirectedPaths.java class in the given Algs4.jar library provided in the starter code,

I found the SAP by first finding the LCA (lowest common ancestor) and then computing the sum of both respective vertices distance to the LCA (using the distTo() method in BFDP).

In order to find the LCA, I used 2 methods within the BFDP class; pathTo() and hasPath(), the former returns an Iterable collection of the vertices from the vertex instance of the BFDP object to the parameter vertex, and the latter checks to see if there is a path from the vertex instance of the BFDP object to the parameter vertex. For pathTo(), I used the path between either v or w (the 2 points of which we had to find the LCA) to the root vertex. Then iterating through every element in the Iterable returned, I checked to see if any of them had a path from the corresponding vertex using hasPath().

Given that the height is H, number of vertices is V, and the number of edges is E, the worst case running time would be O(E+V) as indicated by the comments in the provided code (this would only happen if it was a straight line graph, resulting in a linear search), and the best case running time would be O(1), occurring if there was only one vertex, or if v = w.