Data Structures and Algorithms, Design Justification

A **Trie** was chosen as the main data structure for the implementation of Assignment 2, along with a nodal **singly-linked list**.

Tries are usually a nice tradeoff between running time and memory. Whilst first researching for the appropriate data structures for the implementation of this assignment, **a Hash Table** of Strings as keys and a List of Indices seemed quite enticing (along with its O(1) insertion and search operation complexity), and was infact even started off with. Unfortunately, it was dropped after a while because of implementation/design complexities and of course, collisions.

Tries were then chosen after careful comparison between the three potentially good data structures to implement (*Hash tables, DAFSA and tries*, that is). A Trie was preferred for various reasons:

1. A Trie doesn't need a hashing function, as every key can be represented alphabetically, and is uniquely retrievable. Hence, no collisions to deal with.

2. A huge amount of work in setting a trie up happens at the start. Essentially, all of the memory allocations for each array need to be done when words are being added early on. But over time, since words tend to get repeated, adding intermediate nodes becomes a lot easier for their nodes & references have already been initialised. This serves as a benefit to this particular implementation of a textual search engine, where a huge chunk of words, often repeated, need to be stored for fast look-up.

3. The assignment asked for a prefix search implementation, and tries are famous for their autocompletion and text search, where for a given prefix all of the children nodes that hang off the prefix can be traversed through in linear time.

4. Tries are widely used for storing auxiliary information on top of just words themselves (the index of a word in this case). The leaf node usually stores information. This was the main reason a trie was preferred over a deterministic acyclic finite state automaton (DAFSA) ; it would consume less space than a trie but would be more optimal if the storage of words was all that was needed. A DAFSA can be made to directly store auxiliary information relating to each of its potential paths, but a trie was simpler, more appropriate and more graceful for this implementation.

Hence, it was clear that the worst case performance of a trie would be better than a rather poor implementation of a hash table or a DAFSA. The indices of the words would be stored inside a LinkedList in the leaf node of every word (ie, every word will contain a LinkedList of its position in the file). Moreover, the insertion and searching for a trie run in linear time, directly proportional to the length of the words that its storing, allowing efficient setup and search (given that we're not looking for words such as *supercalifragilisticexpialidocious*).

However, tries take up a fair amount of space with their empty reference pointers. With a worst-case space complexity of O(M*N), where M is the average length of all strings in the trie and N being the number of strings, tries can prove to be really inefficient if there is a lack of common prefixes or repeated words. But since this possibility is ruled out in our implementation, tries with linked lists storing indices of words seemed to be an appropriate choice. Further improvements could include compressing the current implementation of a trie to reduce memory consumption and take advantage of the same functional benefits of a trie.

For other purposes in the implementation, such as data manipulations, storing results etc, Linked Lists were used. They were a preferred data structure for they can grow and shrink at runtime, and since random traversals were not needed for the implementation. Moreover, insertions for the scope of the implementation were O(1) (Always inserted at the back/front, since order isn't necessary), and deletions happened in linear time. Therefore, a Linked List was deemed to be an appropriate data structure for the context.

References/Important Links Used

1. https://en.wikipedia.org/wiki/Trie
2. https://www.cs.cmu.edu/~fp/courses/15122-f10/lectures/18-tries.pdf
3. http://www.bradcypert.com/2017/06/30/a-brief-introduction-to-tries/
4. https://www.sanfoundry.com/java-program-implement-singly-linked-list/
5. http://www-poleia.lip6.fr/~constantin/grbd/sellis.pdf
6. https://medium.com/basecs/trying-to-understand-tries-3ec6bede0014
7. http://www.ardendertat.com/2011/05/30/how-to-implement-a-search-engine-part-1-create-index/
8. https://nlp.stanford.edu/IR-book/html/htmledition/a-first-take-at-building-an-inverted-index-1.html
9. https://www.quora.com/Can-Trie-prefix-tree-be-used-to-build-a-mini-search-engine-on-my-local-machine
10. https://www.quora.com/What-is-the-best-way-to-implement-a-Trie-in-Java