CSCI203 - Assignment One Report

Author: Dinh Che **Login:** dbac496 **Student ID:** 5721970 **Email:** dbac496@uowmail.edu.au

Overall Solution Strategy

1. Either a filename is passed as an argument to the program or a prompt is presented to ask for the filename.
2. The application opens the file for read and reads one word at a time which is converted to a char array.
3. The word is put through a `preprocess_word(char^)` function that uses standard library functions `isalpha(char)` and `tolower(char)` to remove non-alpha characters and converts it to lowercases respectively.
4. A binary search function, `search(Node^, char^)` is used to search the `struct Node^ WORD_TREE` to find an instance of the word. If an existing occurrence is found, it returns an index `key` to that word in the `WORDS[]` array which is then accessed to increase the count.
5. If no instance of the word was found, a new `struct Word word_struct` is created with the `start_idx` set as `NEXT_CHAR`. The word is then added to the `POOL[]` array and the `word_struct.length` is incremented. The `word_struct` is then added to the `WORDS[]` array and inserted into the `Node ^WORD_TREE` AVL tree.
    1. During insertion into the AVL tree, memory was allocated in the `insert(Node^, int)` function because to allow this `Node` to escape the scope of the function required putting it in the heap memory rather than the stack for the function.
    2. A custom `word_str_cmp(Word^, Word^)` was used to place the `Node` in its correct position in the tree.
    3. The `word_str_cmp(Word^, Word^)` uses the `start_idx` in the `Struct Word` to compare chars in the `POOL[]` string pool.
6. Once all words have been inserted the `WORDS[]` array is sorted with a quicksort algorithm. The `quicksort(Word[], int, int)` partitions the array based on a pivot chosen as the last element. The comparison made for sorting include the count and alphabetical ordering using the `compare(Word^, Word^)` function.

Data Structures Used

- `Word` structure - a structure to store some information about each word with respect to its placement in the string pool and its count. It has the following properties:
    - `int start_idx` - the index value where the word begins in the string pool POOL[].
    - `int length` - the length of the word.
    - `int count` - the number of occurences of the word from reading in.
- `Node` structure - a structure with the following properties for an AVL tree.

    - `int key` - the key represented as an index for the `word_struct` in the `WORDS[]` struct array. To reduce the size of the `Node`, an `int` was used to store an index to the `word_struct` where you could access to get the `word_struct.start_idx` to do comparisons for insertion operations. This decision was made to save on memory rather than inserting and balancing a larger-sized node.
    - `int height` - the height of the node in the tree.
    - `Node ^left, ^right` - pointers to the left and right child of the `Node`. Pointers were chosen because an array-based implementation of AVL trees are costly even despite the tradeoff with some dynamic memory for initialising the `struct Node` in the heap.
- `POOL[]` - A string pool array for compactly storing chars from the words read in.
    - A string pool `POOL[]` is used to maintain storage of the words in memory compactly. This also avoids having to store them as part of the AVL tree or the Word struct array. Accessing the characters is fast as arrays can be accessed instantly with `O(1)` time, however, the time taken to read or compare the word would require $O(length)$ which is still relatively fast compared to the cost of moving larger strings around.
- `WORDS[]` - An array to store the Word structs as they are read in.
    - Accessing the `WORDS[]` array via index to increase a `struct Word` count has a complexity of $O(1)$.
- `Node^ WORD_TREE` AVL tree - an AVL tree to store the words as they are read in.
    - Binary search with a balanced AVL tree has a complexity of $O(logn)$ with a successful search limited by the height h and an unsuccessful search very close to h.
    - Insertion into an AVL tree has an upper bound of $O(logn)$ after every insertion and for rebalancing. Coupled with the efficiency for searching an AVL tree, this proved the best tradeoff choice.

Standard Algorithms Used

- AVL Trees with Struct pointers.
- Quicksort with last element as pivot.
    - Quicksort was chosen because it sorted in-place on an array with an average time complexity of $O(n \cdot logn)$.