Programmer's Documentation

1. Specification Breakdown

- The program loads a dictionary from either the console or a text file. Then, upon entering a word, it searches the dictionary for any anagrams and lists them off. A warning is listed instead if no such word was found. More details: Assignment.pdf
- There are 3 functional requirements:
    1) The ability to input words of set maximal length through console.
    2) The ability to input words of set maximal length through a text file.
    3) The ability to search the dictionary with reasonable speed.

2. Architecture / Design

- The program is divided into 4 stages:
    1) The initial menu
    2) Console/Textfile input (maps to FR 1, 2)
    3) Unscrambling (maps to FR 3)
    4) The end menu
- The code is split into 2 procedures and 1 function:
    1) Procedure `InsertIntoTable(…)` maps to FR 1, 2
    2) Function `Hash(…)` maps to FR 1, 2
    3) Procedure `WriteList()` is for testing purposes only
- Data structures are as follows:
    1) Linked list of `PNodes`
    2) Array of linked lists declared as `MainArray`
    3) (1) and (2) are used as a Hash table (maps to FR 1, 2, 3)
- Global variables:
    1) `Node1`, `Node2` are used to search and expand the linked list
    2) `Load` is used for console input
    3) `Index` is used to access the correct linked list in the `MainArray`
    4) `i`, `n` are both used in for cycles
    5) `sum` is used in the Hash function to hold the sum of ASCII values
    6) `token`, avail, fail are search flags
    7) `TextFile` is used for every operation with a text file
- The program takes advantage of these algorithms:
    1) The hashing algorithm takes a `string` as an argument and returns an `index` from 0 to 99 (0 to `SizeOf(MainArray)`), that is used to access the correct field in `MainArray`. The algorithm must be very fast, simple, yet functional with every `string` input. It does not use any extra memory and the time complexity is O(n), where n equals Length(`string`).
    2) Hash table Insert algorithm is used to find a correct place for each word and resolve any collisions. In this case, collision resolution is implemented as separated chaining. The algorithm works with the `index` value and searches `MainArray[index]`. If full, it creates a new node and connects it to the end of `MainArray[index]` linked list. On average insertion takes O(1), in the worst case O(n), where n is the length of the said linked list.
    3) Hash table Search algorithm can find any value in the hash table in constant time (O(1)), in the worst case linearly (O(n), where n is the

length of the `MainArray[index]` linked list). It access the index generated by `Hash(…)` function in `MainArray` and reads the value (values, if a collision occurred) stored there.

4) Words are compares using a simple algorithm, that checks each character in the first string and ensures, that it is present in the second string. Lengths of both strings are compared at the start. This, paired with equal sum of ASCII values, ensures it is correct. Time complexity is $O(n^2)$.

- The user input is loaded into the `Load` string in every scenario. In stages 1 and 4, it is simply compared to '0', '1', and '2'. `Load` also serves as the argument for the `Hash(…)` and concatenated with '.txt' for `AssignFile(…)`.

3. Technical documentation

- Hash Table is the essential part of this program. Every string that is being loaded into the table has a special value, based on which the index is selected. In this case, it is the sum of ASCII values of every character in the string mod 100 (`SizeOf(MainArray)`). However, this doesn't guarantee that every value has a unique index, since it is quite easy to create two or more different strings with the same sum mod 100. Additional values therefore need to be saved in a linked list – this is called collision resolution via separate chaining. Each field in `MainArray`, `MainArray` being the body of this hash table, is a pointer (`PNode`) and a head of a linked list. In short, `MainArray` contains heads of linked lists. Each Node contains `Value` (string) and Next (`PNode`). Every `Value` is a single entry of the dictionary. Reading from and writing into this table requires basic knowledge of pointer operations.
- Function `Hash` has a single input value of the `string` type and returns `byte`. It calculates the output using following formula.

$$output = \left(\sum_{i=1}^{length(input)} ord(input[i]) \right) \% \, SizeOf\,(MainArray)$$

- Procedure `InsertIntoTable` takes a `string` and a `byte` as arguments and traverses the linked list of `MainArray[byte]` until it reaches the end, where it creates a new node, saves the `string` input into it and ties it with the `Next` pointer of the last visited node. If no nodes other than the head exist, then the procedure inserts the `Value` the head of the list.
- Procedure `WriteList` traverses the `MainArray` and stops on each index to look for additional nodes. It searches these nodes until it reaches the end, printing the `Value` of each node with associated index into the console.