Lab 2 Design Document

1. Class Designs

1.1 HashTable Class

Description: My HashTable class is called when a new hash table of size m is created.
Constructor *HashTable(int m):* My HashTable constructor takes an integer parameter which represents the size of the hash table object created. It will then create an array of linked lists using the LinkedList class to handle collisions and also initializes the dictionary to hold all the words.
Destructor ~*HashTable():* When the destructor is called, all memory is deallocated by deleting each linked list from hash table, the hash table array and the dictionary.
Private Member Variables:

- *int hash_size:*This variable stores the size of the hash table (int m)
- *Dictionary *new_dictionary:* A pointer variable pointing to an instance of a Dictionary object
- *LinkedList **new_hashtable;* A pointer variable pointing to pointers of instances of LinkedList objects

Member Functions:

1. *int getHashKey(string word):* This function will add all the ASCII values of each character of the given word from the parameter and return the hash key
2. *bool insert (string word):* This function allows you to insert a word into the dictionary and the hash table after ensuring that the word is only composed of alphabetic characters and is not a duplicate.
3. *bool tokenizeFile(fstream& fin):* This function reads a file, iterating through each word and adds all the new eligible words into the dictionary and hash table while also keeping track of the number of words added.
4. *void getToken(string word):*This function retrieves the token associated with the given *word* parameter by getting the hash key of the word and then going to the associated bucket in the hash table and search for the word.
5. *void getWord(int token):* This function retrieves the word associated with the given *token* parameter by returning the word at that index in the dictionary array
6. *void printKeys(int k):* This function iterates through the entire linked list and prints the key of each word
7. *bool allAlphabetic(string word):* This function iterates through each character in the given *word* parameter and checks if they are all alphabetic.l

1.2 Dictionary Class

Description: The Dictionary is a dictionary that stores all of the words.
Constructor *Dictionary(int m):* My Dictionary class takes one integer parameter, which represents the beginning size of the object, which will be the same size as what was chosen for the hash table. It will then initialize the *capacity* variable to the *m* parameter and initialize the *size* to 0 as the dictionary will be empty for now. The **dictionaryArray* pointer will be set to an empty array of strings with size *m*.
Destructor ~*Dictionary():* The destructor will properly deallocate all the memory by deleting the string array.
Private Member Variables:

- *int size:* This variable stores the current size of the dictionary
- *int capacity:* This variable stores the max capacity of the dictionary
- *string *dictionaryArray:* A pointer variable of type String which will store all the words in an array of strings

Member Functions:

1. *void addWord(string word):* This function will add the given *word* parameter to the end of dictionaryArray and increase the *size* variable by 1. If size exceeds the capacity of the dictionary then it will create another array with size of "capacity+50" and copy the old array into the new bigger array
2. *bool duplicate (string word):*This function iterates through dictionaryArray and checks if the given *word* parameter is the same as a word in the dictionary
3. *int get_index():*This is a getter function for the current size of the dictionary
4. *string get_word(int token):* Returns the word in the dictionary located in the index given from the *token* parameter

1.3 LinkedList Class

Description: My LinkedList class is my linked list class. This class is called and used when the hash table is created. A new linked list object is created for each bucket of the hash table and stores all the Word objects with identical keys.
Constructor *LinkedList():* My LinkedList constructor initializes the **head_word* pointer to nullptr as the linked list will start off as empty
Destructor ~*LinkedList():* When the "EXIT" command is called, the destructor will get called as well. My destructor iterates through the linked list, deletes each Word object and sets their pointers to nullptr. This will free up the memory used by the list and prevent and memory leaks.
Private Member Variables:

- *Word *head_word:* This is a pointer variable representing the beginning of the linked list

Member Functions:

1. *void addWordinHash (string word, int index):* This function iterates through the linked list, creates a new Word object with the given *word* and *index* parameters and inserts it to the end of the list
2. *int getToken (string word):* This function iterates through the linked list and finds if there is a Word object that has the same word and the given *word* parameter
3. *Word *get_head_word():* This is a getter function and provides access to the *head_word* pointer

1.4 Word Class

Description: This is my node class, which is called and used whenever a new word is added to the hash table. Each word represents a node in the linked list.

Constructor *Word(string word, int index):* My Word constructor takes a string parameter and an integer parameter and is called when a word is added to the hash table. This allows the class to store the word and its index in the dictionary for each object of Word. It will then initialize the *current_word* and *current_index* member variables with the respective parameters. It will then set the **next_word* pointer to nullptr because the new node will not connect to the linked list yet.

Destructor ~*Word():* When the "EXIT" command is called, the destructor will get called as well. It will set *current_word* and *current_index* to "" and 0 respectively and **next_word* will be nullptr. This ensures that the object is properly finalized before its memory is deallocated.

Private Member Variables:

- ***string current_word***: This variable stores the word
- ***int current_index:*** This variable stores the index of that word in the dictionary
- ***Word *next_word***: This variable is a pointer which allows a Word object to point to another object of Word as its "*next_word*", creating a chain of linked objects

Member Functions:

1. *string get_word():* This is a getter function for the word and provides access to the private variable
2. *int get_index():* This is a getter function for the index and provides access to the private variable
3. *Word *get_next():* This is a getter function for **next_word* and provides access to the private pointer
4. void set_next (Word *new_next_word): This is a setter function for the **next_word* which allows you to point the current Word object to another Word object given in the parameter, connecting the nodes in a linked list

2. Expected Runtime

When calculating expected runtimes, uniform hashing is assumed. This means that all the elements should be evenly distributed across the slots in the table with little to no collisions.

2.1 "TOKENIZE" and "RETRIEVE" commands

The average runtime of these two commands is O(1) because, assuming uniform hashing, these functions perform a constant number of operations that do not depend on which slot you are in.

2.2 "INSERT" command

The average runtime of this command is O(1). This is because, assuming uniform hashing, there are no or very few collisions. Thus, the function performs a constant number of operations regardless of size. However for the worst case, where there are a lot of collisions, you would have to traverse the linked list once and perform a O(1) operation at each node.

3. UML Design