PART I

The GUI is to be started from MyLanguageModel.java file and is to be executed using the following commands

* javac MyLanguageModel.java
* java MyLanguageModel

The User Interface has the following areas and their uses:

1. Document Area: The Document/Text which is to be loaded from the file will appear.
2. Load Document Button: On pressing this button, A file chooser is prompted where the user can locate the text file and load it by browsing the directories.
3. Process document Button: On pressing this button, the document text is inserted, into hash table, the text is processed and showcased in the table according to the occurrence.
4. Sort Unigrams: Order unigrams in descending order of frequency.
5. Load Bigram Button, Load Trigrams: On pressing these button, respective Hash Tables are created and words are inserted based on the hashing scheme.
6. Show Statistics: Showcase the Total words, different words, Avg List size, Avg Std Deviation etc.
7. Predict Next: Predict the next sequence of words, by calculating probability.
8. Hash Unicode Sum: Switch to Unicode sum hashing technique for inserting words in hash table.
9. Hash First Letter: Switch to First Letter sum hashing technique for inserting words in hash table.

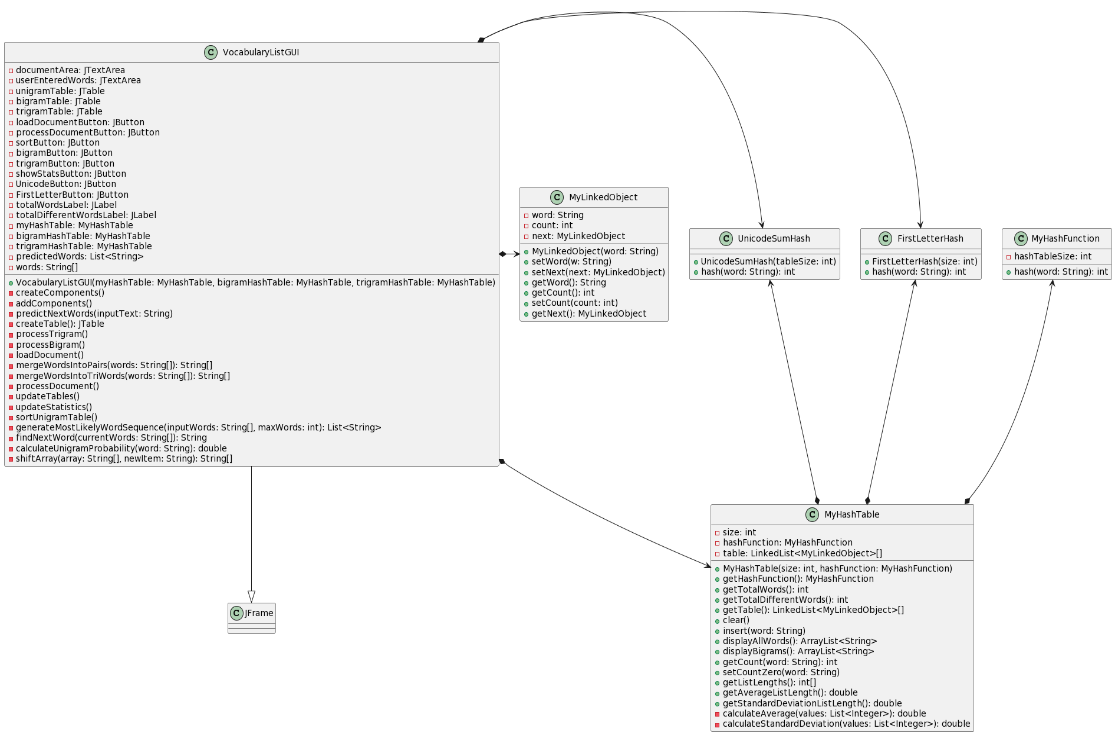
**Statistics:**

1. Total words: This label displays the total words processed when Process button is clicked.
2. Total Different words: This label displays the total-different words processed when the Process button is clicked.
3. Average List Length: This label displays the average list length at each index, this metric is used then used to calculate other important performance metrics
4. Standard deviation List Length: This label displays the standard deviation of list length at each index, this metric can be used to understand the performance of the Hash Table structure.

# Part II

1. **Data Abstraction**: The `MyHashTable` class encapsulates the details of hash table operations, providing a higher-level interface for users. The `MyLinkedObject` class abstracts the concept of a linked object with fields like word, count, and next.
2. **Encapsulation**: The fields in `MyLinkedObject` (e.g., word, count, next) are private, encapsulating the internal state. The `MyHashTable` class encapsulates the implementation details of the hash table, and its methods provide an interface to interact with the hash table. Inheritance: In the provided code, there is no explicit use of inheritance. Polymorphism: The `MyHashTable` class has a `MyHashFunction` interface, allowing for different implementations of hash functions.
3. **Collections Framework**: The `MyHashTable` class represents a basic form of a hash table, providing a structure for storing and managing elements. The List interface is used for linked lists in the `MyHashTable` class.
4. **GUI**: The `VocabularyListGUI` class represents a graphical user interface using Swing components. Components like `JTextArea`, `Jtable`, `JButton`, etc., are used to create a GUI for interacting with the hash table and displaying information.
5. **Exception** **Handling**: There is some basic error handling, such as handling file reading errors in the `loadDocument` method of the `VocabularyListGUI` class,
6. **Data** **Structures**: The `MyHashTable` class uses a hash table structure, and the `MyLinkedObject` class represents linked objects within the hash table.

**UML Diagram**



**Collections Framework:**

* Hash Tables: The code uses a custom implementation of a hash table (MyHashTable) to store word occurrences. It leverages the Collections framework to handle the storage and retrieval of words efficiently.
* Lists: The code uses ArrayList to store predicted words (List<String> predictedWords). The ArrayList is part of the Java Collections framework and provides dynamic array functionality.
* Sorting with Collections: The generateMostLikelyWordSequence method utilizes a custom sorting of words based on their probabilities. Collections sort method could be used, but in this case, a custom sorting logic is implemented.

**Graphical User Interface (GUI):**

* Swing Components: The GUI is built using Java Swing components, including JFrame, JTextArea, JTable, JButton, JLabel, and JOptionPane.
* Event Handling: Event listeners are used to handle user actions. For example, ActionListener is implemented for buttons like loadDocumentButton, processDocumentButton, etc. The GUI responds to user input by triggering appropriate methods.
* SwingUtilities.invokeLater: The SwingUtilities.invokeLater is used to ensure that GUI-related tasks are executed on the Event Dispatch Thread (EDT). This is important for thread safety in Swing applications.

**Exception Handling:**

* File Loading Exception Handling: The loadDocument method includes a try-catch block to handle potential IOException when reading a file. It displays an error message using JOptionPane if an exception occurs.
* Error Messages for Empty Documents: In various methods (processDocument, processBigram, processTrigram, sortUnigramTable, updateStatistics), the code checks for specific conditions (e.g., empty document or empty hash tables) and displays error messages using JOptionPane to inform the user.
* Predict Next Words Exception Handling: The predictNextWords method includes a try-catch block to handle any exceptions that might occur during the word prediction process. It displays an error message using JOptionPane if an exception occurs.

**PART III**

**TASK 1**

Task 1 involves the implementation of a class named MyLinkedObject, which represents an object in a linked list. Here's an explanation of the code:

**Attributes:** word: Represents the word associated with the object. count: Represents the count or frequency of occurrences of the word. next: Represents the reference to the next object.

**Constructor:** The constructor MyLinkedObject(String word) initializes the object with a given word. It sets the initial count to 1 since it's the first occurrence of the word. The next reference is set to null.

**setWord Method:** The setWord(String w) method is used to update the linked list based on a new word as per the requirement and word checks. **setNext Method**: The setNext(MyLinkedObject next) method is used to set the reference to the next object. **getWord Method:** The getWord() method returns the word associated with the object. **getCount Method:** The getCount() method returns the count or frequency of occurrences of the word. **getNext Method:** The getNext() method returns the reference to the next object in the linked list.

**TASK 2**

The provided abstract class MyHashFunction serves as a base class for implementing various hash functions. Two specific hash functions, FirstLetterHash and UnicodeSumHash, are mentioned. Let's take a look at their potential implementations:

1. **FirstLetterHash**: This hash function uses the first letter of a word sequence. The hash value is calculated by taking the Unicode value of the first letter modulo the hash table size.
2. **UnicodeSumHash:** This hash function calculates the sum of Unicode values of all characters in a word sequence. The hash value is the sum modulo the hash table size.

**TASK 3**

The design of the MyHashTable class encapsulates instances of MyLinkedObject and MyHashFunction classes, which adheres to the concept of encapsulation. Both MyLinkedObject and MyHashFunction are private inner classes/interfaces within MyHashTable, and they are not directly exposed to the outside world. This encapsulation ensures that the internal details of these classes are hidden and only accessible through the MyHashTable interface.

Here are the reasons why this design achieves encapsulation:

1. MyLinkedObject is private:The MyLinkedObject class is declared as a private inner class of MyHashTable. This means it is only accessible within the MyHashTable class, and its details are not exposed to external classes.
2. MyHashFunction is an Interface:The MyHashFunction interface is declared as a private inner interface of MyHashTable. Like MyLinkedObject, it is not directly accessible from outside classes.
3. Internal Usage:Methods within MyHashTable that utilize MyLinkedObject and MyHashFunction are private or package-private, restricting their use to within the MyHashTable class.

This design choice adheres to the principle of information hiding and encapsulation. The external code that interacts with MyHashTable does not need to be aware of the internal implementations of MyLinkedObject and MyHashFunction, promoting a cleaner and more modular design.

In summary, the design appears to achieve encapsulation by appropriately scoping the visibility of MyLinkedObject and MyHashFunction within the MyHashTable class.

**Task 4**

Here's a general strategy for ordering vocabulary list:

1. Retrieve the Frequency Information: Iterate through your vocabulary list. For each word, retrieve its frequency information. This might involve looking up counts in your data structure (e.g., MyHashTable).
2. Sort the List: Use a sorting algorithm to sort the vocabulary list based on the frequency of each word. When sorting, arrange words in descending order, so the word with the highest frequency comes first. I have used a comparator to compare the frequencies in the tablemodel.
3. Create a New List: Create a new list or update the existing list with the sorted order.
4. Observations from the Vocabulary Lists:

High-Frequency Words: Words that appear frequently in the text will be at the beginning of the list.

Low-Frequency Words: Words that appear infrequently will be towards the end of the list.

**Statistics for lengths of individual linked lists within the hash table using Hash Table size(m=10)**

|  |  |  |
| --- | --- | --- |
|  | Lengths using Unicode Sum Hashing | Lengths using First Letter Hashing |
| 1 | 1723 | 1430 |
| 2 | 1773 | 1269 |
| 3 | 1790 | 2086 |
| 4 | 1741 | 602 |
| 5 | 1786 | 1796 |
| 6 | 1793 | 2556 |
| 7 | 1749 | 1097 |
| 8 | 1784 | 1647 |
| 9 | 1678 | 1839 |
| 10 | 1733 | 3228 |

|  |  |  |
| --- | --- | --- |
| Standard Deviation | 35.3 | 710.3 |

**Unicode Sum Hashing:** Observation: First value is 35.3. Unicode sum hashing involves summing the Unicode values of characters in a string. This method is simple and may work well for certain applications. However, the summing operation can lead to collisions (different strings producing the same sum), especially when the strings are of different lengths or have different character distributions.

**First Letter Hashing:** Observation: Second value is 710.3. First letter hashing involves using the Unicode value of the first letter of the string as the hash value. This method is also simple but may suffer from collisions if many strings start with the same letter.

**Observations on Provided Values:** The first value (35.3) for Unicode Sum Hashing seems quite low, and it may indicate that the summing operation didn't result in a well-distributed hash. The second value (710.3) for First Letter Hashing is relatively higher, but it's essential to consider the nature of your data. If many strings start with the same letter, this method might result in more collisions.

**Benefits of sorting ‘linked lists’ by the decreasing number of word occurrences, instead of ones ordered alphabetically:** Sorting linked lists by the decreasing number of word occurrences (frequency) instead of alphabetically offers benefits such as improved relevance and enhanced predictive power. This approach facilitates faster access to more meaningful and contextually relevant words, making it advantageous for natural language processing tasks, statistical analysis, and tailored language models.

**Task 5**

When calculating the probability using unigrams (individual word probabilities), the expression simplifies to: p(w1, w2 … wk) = p(w1).p(w2)…p(wk)

In this case, the probability p(w1) is the probability of the first word in the sequence. It represents the likelihood of the occurrence of the first word independently.

**Calculate Unigram Probabilities:**

The probability p(w1) is obtained by dividing the count of occurrences of the word w1 ​ by the total number of words in the corpus. This gives you the probability of w1​ occurring independently.

Multiply Unigram Probabilities:

To calculate the overall probability of the entire sequence, you multiply the individual unigram probabilities together. This assumes that the occurrence of each word is independent of the others.