Colby Wirth
Professor Behrooz Mansouri
24 November 2024

An Analysis of the Implementation of a Hash Map and Sentiment Analysis Model

File Input Functions

The input function, MyDataReader.readDataToHashMap, reads the input files line by line. From each line a custom Tweet object is instantiated. This is done in $O(4n)$ as there are four attributes per tweet. This simplifies to $O(n)$.

MyHashMap Implementation Overview

Constructor

The current implementation only has one constructor. In this constructor, an empty hash map in instantiated with null values. Therefore, it operates in constant time, $O(1)$.

MyHashMap.put

The *put* method inserts an element into the hash map's table. Each element is hashed by java's hashing function in constant time, then it is inserted into the table based upon the hash value. When elements have unique hashes this is performed in $O(1)$ time complexity. When elements have duplicate hashes, collision operates are performed. In this implementation, this is solved by chaining with linked lists. The chaining operations involve searching the linked list located at the index for a matching key. If no key is found, the element is inserted at the tail of the linked list. This operation performs at $O(k)$, time complexity, where k is the number of elements in the linked list and $k \ll n$. This simplifies to $O(1)$ time complexity.

When the current size of the hash map's table reaches the threshold value, (75% of the table's capacity), the *MyHashMap.resize* method is called at the end of the put method. This method operates by iterating over every element in the hash map. This is done in $O(n)$ time. This is the worse case time complexity for the put method.

MyHashMap.get

The *get* method retrieves an element from the hash map's table. The searching element is hashed and its location is found

in *O(1)* when elements have unique hashes as mentioned before. As with the *put* method, when elements have duplicate hashes, an inner linked list must be traversed to find the element. This is done in *O(k)*, where $k \ll n$. Therefore these operation is performed in *O(1)* time complexity.

MyHashMap.keySet

I implemented this method to better aid operations. The keySet method returns a prebuilt hash set of all keys, *K*. This set is updated as each new key is added to the hash map; therefore, the keySet method only needs to return the reference to the prebuilt set. This is done in constant time.

MySentimentAnalysisModel Implementation Overview

Constructor

The constructor for the sentiment analysis model is built by iterating through each key in the prebuilt hash map of training data. The constructor calls the method: *addTweetToModel* once for each of *n* elements in the training data's key set. This method handles the logic for cleaning the data in each tweet. This helper method iterates on each word once. Therefore, the time complexity of the constructor is *O(nk)*, where *k* is the total number of words across all tweets. This simplifies to *O(n)*.

MySentimentAnalysisModel.testModel

The testing method iterates on all *n* tweets in the testing data set once. For each tweet, a helper method *sentimentPredictor* is called. This helper method performs the cleaning and categorization for all *k* words. Therefore, the testing method executes in *O(nk)* time, which simplifies to *O(n)*.

Output Results from the Data

1732 milliseconds to build the train hash map
17 resizing to build the train hash map
81 milliseconds to build the test hash map
13 resizing to build the test hash map
Ratio of correct predictions: 0.6273125

Note, the ratio of corrected predictions (aka accuracy score) exceeds the specified threshold of *0.50*. Therefore, it is considered to be successful for
this project.