**Project 6 Report**

**Name: Yuyang Wang**

**Course: CS 231**

**Section: A**

**Date: 4/17/2023**

**Title:** Checking word frequency using BST map and Hash map

**Abstract**:

The core concept of this project is to use different implementations of Mapset to check the word frequency of a large text document. In this project, the implementations are Hashmap and BSTmap. The program first reads the text file and extracts all the words in the document using REGEX. Then it will build the map base on these words. Then the output file would be written with words and its value(times appearing). Also when analyzing these words, common words are ignored. The goal of this project is to analyze the most frequently used words throughout the year, and the time it would take to build the map using different structures. The program can demonstrate the total words, unique words, time taken to build the map, and the most frequently used word in that year.

**Exploration:**

1: Analyzing the frequency of the words each year. The common words ignored here are words provided by the professor and words that were added, which would appear to be highly frequently used every year.

Table 1: The most frequent words each year. (The output of MainHash.java)

| Year | 2008 | 2009 | 2010 | 2011 | 2012 | 2013 | 2014 | 2015 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Most frequent word | better | thing | thing | always | look | look | game | game |

From these words, “always” might point to users discussing habits or ongoing trends. The word "look" in 2012 and 2013 might imply that users were giving more attention to visual aspects, possibly related to the rise of image and video sharing on the platform. The prominence of "game" in 2014 and 2015 could suggest that gaming-related discussions were particularly popular during those years. This makes sense, as online gaming and eSports saw significant growth around that time. Also, Nintendo 3DS released at that time.

Interestingly, a friend told me that the word “obama” is frequently used. It’s pretty interesting to look at the frequency change of it year by year.

Table 2: The word “obama” appearing times and its appearing frequency. (The output of MainHash.java)

| Year | 2008 | 2009 | 2010 | 2011 | 2012 | 2013 | 2014 | 2015 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| The word "obama" appearing times | 11407 | 6974 | 3866 | 3414 | 2550 | 1642 | 904 | 829 |
| The word "obama" appearing frequency | 0.0024 | 0.0006 | 0.00029 | 0.00028 | 0.00022 | 0.00014 | 0.00007 | 0.0000069 |

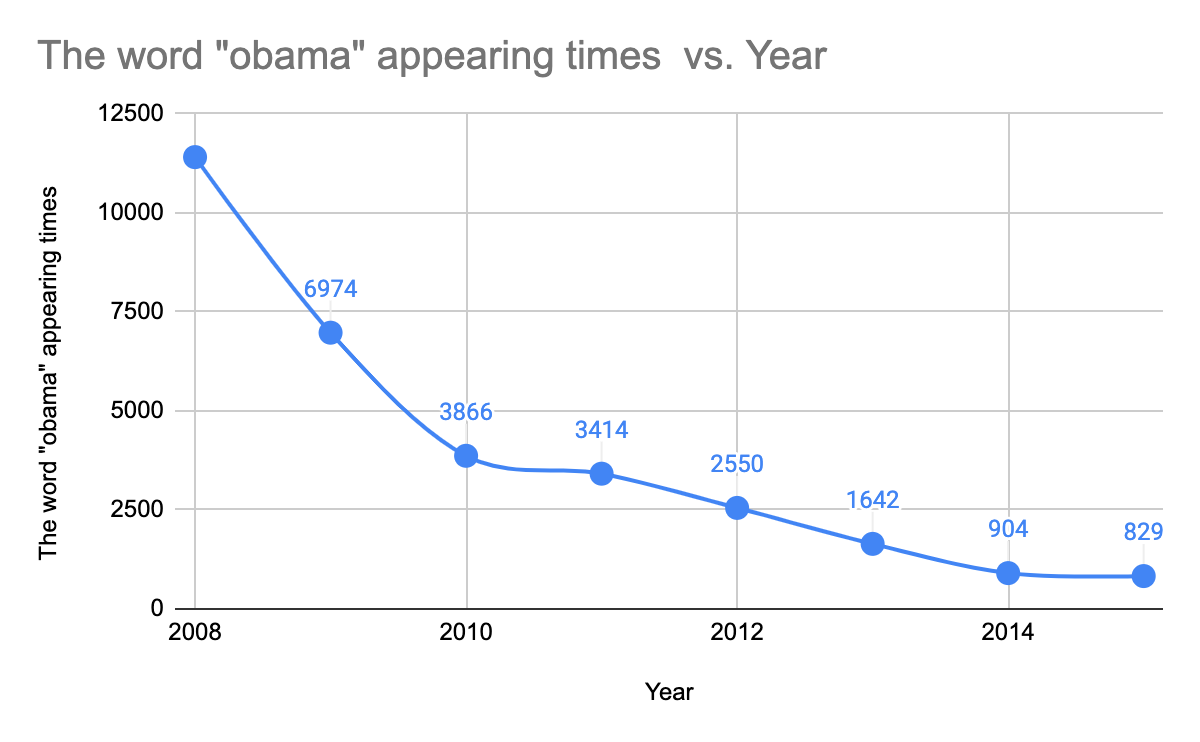


Figure 1: The appearing times of the word “obama”

This can be explained. 2008: Barack Obama was elected as the 44th President of the United States in November 2008. As a result, discussions about him and his policies were likely at their peak during this year. 2009: Obama's inauguration took place in January 2009, and this was the first year of his presidency. Although the frequency of "Obama" decreased compared to 2008, it still remained relatively high due to ongoing discussions about his administration. 2010-2011: During these years, Obama was still serving his first term as president. The frequency of his name continued to decline, which could be attributed to the natural decrease in novelty and interest over time. 2012: This year marked the U.S. presidential election, in which Obama was re-elected for a second term. The appearing times are not dropping significantly, as from 2008-2010. 2013-2015: These years were part of Obama's second term as president. As his term progressed, discussions about him and his policies likely became less frequent, leading to the continuous decline in the frequency of his name.

2: Analyze the total time it takes for different data structures to build the map.

Take three runs for each year and gain average results of the time taken for building different maps.

Table 3: The time taken for building maps of different structures. (The output of MainHash.java and MainBST.java)

| Year | 2008 | 2009 | 2010 | 2011 | 2012 | 2013 | 2014 | 2015 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Time taken for hash map (ms) | 2519 | 6684 | 8880 | 8671 | 7162 | 5647 | 6125 | 6273 |
| Time taken for BST map (ms) | 15722 | 29077 | 24712 | 24115 | 21329 | 18903 | 20539 | 24167 |
| Total words | 12308487 | 30006889 | 34863964 | 32166119 | 29740218 | 28944670 | 30513654 | 30259997 |

Table 3 shows that the time taken for building the hashmap is much faster than building the BST. Also, the relationship between building time and input data size is related. With the increase of the input data size, the time taken to build the map would increase.

3: Analyze the max depth of the data structures after calling the buildMap method.

Table 4: the max depth of hash map and BST map (The output of MainHash.java and MainBST.java)

| Year | 2008 | 2009 | 2010 | 2011 | 2012 | 2013 | 2014 | 2015 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Max depth of hash map | 5 | 7 | 6 | 7 | 6 | 6 | 6 | 6 |
| Max depth of BST map | 77 | 45 | 44 | 43 | 47 | 43 | 44 | 49 |

Table 4 shows the max depth of hash map and BST map. It is obvious that the max depth for hash map is more consistent than the BST map. The reason why hash maps tend to have a more consistent max depth compared to BSTs is because of the hashing function, which ideally distributes the elements evenly across the buckets. This leads to a lower probability of having a very deep structure. In contrast, BSTs are more sensitive to the order of the input data, which can lead to highly unbalanced trees with large max depths. The max depth for a hash map with separate chaining is related to the number of elements, the number of buckets, and the quality of the hash function, while the max depth for a BST depends on the order of the input data and the structure of the tree. Hash maps tend to have more consistent max depths because of the hashing function, while BSTs can have a much wider range of max depths depending on the input data.

**Extensions：**

**Ex1: Building an AVL tree**

To build a self-balancing tree, the put method inserts a key-value pair into the tree. If the tree is empty, it creates a new root node. Otherwise, it calls the put method recursively to insert the node in the appropriate position in the tree. After inserting the node, the updateHeight method updates the height of the node, and the balance method checks if the node is balanced. A node is balanced if the height difference between its left and right subtree is at most 1. The getBalanceFactor method calculates the balance factor of a node as the difference between the heights of its left and right subtrees. The balance method checks the balance factor of a node and performs rotations if needed to maintain balance. If the balance factor is greater than 1, it means the left subtree is heavier, and a right rotation is needed. If the balance factor is less than -1, the right subtree is heavier, and a left rotation is needed. There are four possible cases for rotations: left-left, left-right, right-left, and right-right. The leftRotate and rightRotate methods handle these cases.

Table 5: The number of Unbalanced node and max depth for AVL tree and BST tree. (The output of MainAVL.java and MainBST.java)

| Year | 2008 | 2009 | 2010 | 2011 | 2012 | 2013 | 2014 | 2015 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| number of the unbalanced node for AVL tree | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| number of the unbalanced node for BST tree | 20154 | 34467 | 37324 | 37500 | 38762 | 40874 | 45453 | 50474 |
| Max Depth for AVL tree | 20 | 21 | 21 | 21 | 21 | 21 | 21 | 21 |
| Max Depth for BST tree | 77 | 45 | 44 | 43 | 47 | 43 | 44 | 49 |

The unbalanced node is to have a balance factor that is greater than 1 or smaller than -1.

Table 5 shows that the BST tree is dramatically unbalanced. The AVL tree has the advantage of having a perfect consistent search time of O(log n). However, it also has disadvantages in that AVL trees require additional logic to maintain balance. This includes rotations and additional storage for balance factors, which make the building map time increase. The time taken for building the AVL tree map are all over 40000ms, which is slower than the BST tree.

**Ex2: Using an ArrayList-based map to compare the processing time.**

First, write an Arraylist Map to have put and get methods. Then put 10000 elements to ArrayList map and also AVL map to measure the time. Also to measure the time to get 10000 elements from the ArrayList map and AVL map.

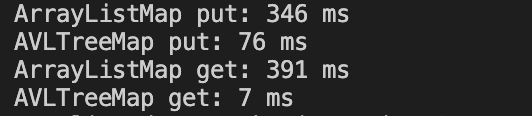


Figure 2: The time taken for putting and getting the elements. (output of MapComparison.java)

Figure 2 shows that the time to put the elements into the ArrayList map and get the elements from it is much slower than the AVL map.

**Ex3: Interesting fact to demonstrate**

To implement the above idea compare AVL tree map and BST tree map. One can find that the put time would be much slower for AVL tree but the get time for AVL tree is faster.

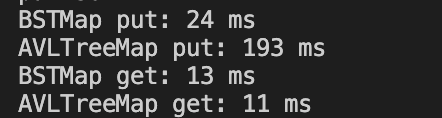


Figure 3: The time taken for putting and getting the elements. (output of AVLvsBSTComparison.java)

**Reflections:**

a HashMap provides faster average-case performance for insertion, deletion, and search operations, but does not maintain any specific order of the keys. In contrast, a BSTMap maintains the keys in sorted order, allowing for efficient in-order traversal, but with potentially slower insertion, deletion, and search operations, especially when the tree becomes unbalanced.

When we use the entrySet method to write your word count files, we iterate through the key-value pairs in the map in a specific order. If we try to read in a large file that was written by a BSTMap, the insertion order of the keys might cause the tree to become highly unbalanced, which can lead to slow performance and, in extreme cases, a StackOverflowError due to deep recursion in the tree. An AVL tree, however, would avoid this issue by self-balancing during insertion, ensuring consistent performance and avoiding excessive recursion.

Reference:

1. <https://www.kaggle.com/datasets/rtatman/english-word-frequency>
2. Ariana He, personal communication 4/16/2023, Colby College.