**How common is your dog's name?**
**Team Comp-slayers**
**Sanika Kurahatti, Tyla Hart**
**Video: https://youtu.be/yN-Cn9t0GZI**
**GitHub link: https://github.com/SanikaKura/DSA-Final-Project**

**Proposal:**

The purpose of this project is to see how common dog names are and what their rank is based on their popularity compared to other dog names. The motivation for this project is for the amusement and entertainment of the statistics for dog owners, future dog owners, or anyone who is curious. The entertainment extends to testing users' own names into the program! The features of the code include a hashmap (an unordered_map) and trie (a 26-ary tree) that use the data from the csv file, store them, and return how often certain dog names appear in the file as well as their rank. Aside from this, the data structures also have a function that returns the top ten dog names. The data used in this project comes from a csv file holding around five-hundred-thousand registered dog names in New York from 2016-2022. The csv file included each dog's name, gender, breed name, and zip code, but for this project, only the names were considered.

In the program, we used the simple and fast multimedia library (SFML) for our graphics as our front end. We also utilized c++ as our coding language and our IDE was CLion. On the matter of algorithms, we implemented the preorder traversal for the rank_traversal and the getTenHelper function, all within the trie header. Each leaf node held the integer value of how many times each name was seen in the csv file, which was the data the preorder traversal through the trie was retrieving. In regards to the hashmap, it was easier to access and iterate through data so it did not take as complex of an algorithm as the trie. One of the additional data structures used was a vector that held the top ten dog names. This was not part of the comparison between the trie and the hashmap, but was merely to entertain and surprise users. The vector began with ten names that we knew only appeared once, and as the trie was traversed, the names with higher counts replaced the old names.

Tyla was responsible for reading the csv file and inserting the large amounts of data into both the hash map and trie. She also developed the data structures to return the amount of times a given name was seen in each structure. Sanika created the traversal of the trie and used it to return the rank of the inputted dog name. She also

used the trie to retrieve the top ten dog names appearing the csv file. The responsibilities of the SFML implementation were split between the two group members as well as the report.

**Analysis:**

Big O Complexities

*Trie*

The header we built to implement our trie contains five key functions which will be analyzed in this section. Starting from the beginning, our insert function has a constant time complexity of O(L), L being the length of the word. This differs from the time complexity of inserting into a binary tree for example, because the location at which a word is inserted into the trie is dependent on each character in the word, rather than the values of the other nodes that were previously inserted. The get_count function has the same O(L) complexity. This is because the characters in a name essentially hold its address, and each node holds the frequency of that dog's name. Therefore, accessing the number of times a name appears is a straightforward process that requires traversing through only the number of nodes that the name has characters.

The traversal function has a O(L*N) time complexity, L being the average length of the words in the tree, N being the number of unique dog names. The get_rank function has the same complexity, because this function performs a traversal while taking count of how many nodes have a higher occurrence.

Lastly, we wanted to include a list of the top ten most common dog names in our dataset. This was achieved using another O(L*N) function. This function does not appear in our hashmap, because it is apart from the main functionality of this project and therefore excluded from the comparison between the two data structures. To return the top 10 dog names, we initialized a vector of ten strings, "TYLA", which aside from being my partner's name, is also the name of one dog in New York. The function then traverses the tree, and each time it comes across a name that is more common than one in our vector, it is inserted and bumps down the other less popular names to a higher index.

*HashMap*

The implementation of our hashmap has three functions that we will be analyzing. First off, the insert function has a time complexity of O(1). This is because the hashing, or determining where in the hashmap to store a piece of data, is a O(1) process. And since we want to keep track of the frequency of dog names, this program

uses chaining to resolve any collisions. Similarly, the get_count function has an O(1) complexity, because it goes through the same process used to insert, but rather than inserting it just returns the value found at that index. Lastly, this ranking function has an O(n) complexity, n being the number of unique dog names. This is because it performs an O(1) operation for each name in our map.

After reviewing the functions of our two data structures, it makes sense that the hashmap had a lower run time compared to the 26-ary tree. The hashmap is more efficient because the insert and get count functions are O(1) operations, as opposed to being a slightly lengthier process for the trie. Tries definitely have certain benefits, such as being more organized when it comes to operations such as traversing the data in alphabetical order, and having a better average complexity for an extremely dense data set. However, for our problem, we were more concerned with the insert and lookup times of names. Therefore, hashmaps are a better choice for our project.

**Reflection:**

This project allowed us to utilize and expand on our previous knowledge and skills regarding data structures, creating a user interface by integrating the sfml library, and developing a unique software project from scratch. Implementing the 26-ary tree and hash map was not something we have done before, but because of our familiarity with similar concepts, it was easy for us to apply our skills we developed from projects such as AVL trees, binary trees, and ordered maps, to the data structures we chose to implement this project.

The most difficult challenge we faced was conceptualizing the 26-ary tree, and brainstorming ideas of how to formulate the ranks of specified user-inputted dog names without using other data structures or causing higher time complexities. We were initially thinking of using a separate data structure, such as a vector or map, to traverse the trie and rank the popularity of all the dog names. However, we realized that the cost of the increased time complexity would be too great, and with time and consideration, we came across a better way to implement this function. Since our tree increments an occurrences value every time a certain name is inputted into the tree, we decided to use our get_count function, which returns this value, and traverse through the tree, while incrementing a count variable which notes each time it comes across a name which had a higher count return value. This count value plus one results in the rank of that name in a highly efficient way.

Apart from the actual backend of this program that reads the csv file and inputs the data into our structures, we decided to implement a user interface for this project. It was a little difficult to use the SFML library to display the results of our program. Though the two of us have used SFML before, we were out of practice and had to quickly figure out and relearn many of the functionalities of the library. Despite this, using SFML was an efficient way to refresh our memory in the respective procedures.'

In terms of splitting our workload and having time management, the two of us didn't have any issues. We did face a little bit of a challenge when the third member dropped this class, because we were only made aware of this in the last week of our project being due. This was a challenge because the two of us had to readjust our responsibilities, and work to rebudget our time in order to get everything done in time.

In retrospect, we would not make any notable changes in the project or workflow, as we used our time wisely, thoughtfully planned how we were going to implement the two data structures, and evenly spread out the work between the members. At most, we could have considered starting the SFML implementation earlier as we were not entirely prepared for its extensive requirement of time. As for what the team members learned as a result of this project: Sanika learned the inner workings of the 26-ary tree and that, particularly, the leaf nodes did not store a string with the value of the inserted word, but that the path of the nodes from the root to the leaf spelled out the characters of the string. The leaf node merely held the boolean value that the word had ended and the amount of time that path had been taken (how many times the dog name was inserted into the tree). Tyla learned the functions used to access a timer and use it to see how long each data structure took to process and manipulate the same data. This included accessing the over five-hundred-thousand data points within the csv file, inputting such data into each data structure, and the implementation of functions such as get_count and get_rank. Using the timer to time the data structures was a new and resourceful way to test the time complexity of the trie and hashmap.

**Data**: https://catalog.data.gov/dataset/nyc-dog-licensing-dataset

**References**:

"Which data structure can be used for efficiently building a word dictionary"
https://www.geeksforgeeks.org/data-structure-dictionary-spell-checker/

"Trying to understand tries"
https://medium.com/basecs/trying-to-understand-tries-3ec6bede0014