**ADS 1 Mid-term CW**

**A poetry assistant**

**Section 1:**

For this assignment, I have chosen to utilize the Dynamic Array and Associative Array (Map) data structures. In section 2, I will provide a detailed explanation of how these data structures are employed. In summary, the combination of these data structures enables the program to achieve remarkably fast lookup speeds when the user searches for rhymes. The Associative Array (Map) allows for constant time complexity (O(1)) when accessing a value by its corresponding key. In contrast, searching an Array typically requires linear time complexity (O(n)).

**Section 2:**

To find and retrieve rhymes efficiently, I utilize an algorithm that gets smarter with each use. First, I import the list of nearly 250,000 words into a dynamic array, ensuring efficient storage and access. When a user types in a word, a special rhyme-finding function gets called.

Once this function is called, it creates a temporary dynamic array data structure. It checks first if the user's word resides in a "rhyme dictionary" – a dedicated associative array (map) storing previously discovered rhymes. If it’s there, it will simply retrieve the word-rhyme pair and assign that to the temporary dynamic array that is then returned to the function caller.

If it is not part of the associative array, thus it is new to the dictionary, it will continue with an iteration over the dynamic array that contains nearly a quarter million words. It will compare the last 3 letters of the user input word with all the words in the dynamic array. If a match is found, it will be added to the temporary dynamic array and it will also add the key-value pair to the associative array (map) for easier and more efficient future retrieval of the rhymes for the same word. Once done, it will return the temporary dynamic array to the function caller.

After a rhyme search, the user will be asked if they wish to continue to type another word or exit the app.

This optimization strategy shines when facing repeat searches. By pre-storing rhymes in the associative array, I avoid needless re-explorations of the dynamic array, ensuring swift rhyme delivery and a seamless, enjoyable user experience for frequent word searches. Finding a value by its corresponding key in the associative array is a fast operation that doesn’t require searching the entire array again for repeated words. The algorithm will get faster the more it will be used, growing the associative array with each new search.

**Section 3:**

dataArray = CREATE\_DYNAMIC\_ARRAY()

rhymesData = CREATE\_ASSOCIATIVE\_ARRAY()

f = OPEN\_FILE(“data.csv”)

READ\_CSV(f, dataArray)

CLOSE\_FILE(f)

CREATE\_READLINE\_INTERFACE(rl)

FUNCTION START\_APP()

FUNCTION GET\_RHYME(word)

rhymesWords = CREATE\_DYNAMIC\_ARRAY()

k = 1

**if** HAS\_KEY(word, rhymesData)

rhymesWords = GET(word, rhymesData)

**else**

**for** i = 1 to LENGTH(dataArray)

**if** LAST\_THREE\_LETTERS(dataArray[i]) is equal to LAST\_THREE\_LETTERS(word)

rhymesWords[k] = dataArray[i]

**END if**

**END for**

ADD(word, rhymesWords, rhymesData)

**END if**

END FUNCTION GET\_RHYME()

PRINT prompt providing the total number of words in database LENGTH(dataArray)

PRINT “Enter a word to find a rhyme: “

word = TO\_LOWERCASE(GET\_USER\_INPUT(rl))

rhymes = GET\_RHYME(word)

**for** i = 1 to LENGTH(rhymes)

PRINT rhymes[i]

**END for**

PRINT prompt informing the number of rhymes found are LENGTH(rhymes)

PRINT “Do you want to type another word?”

answer = TO\_LOWERCASE(GET\_USER\_INPUT(rl))

**if** answer is equal to yes

START\_APP()

**else**

PRINT “Do you want to see the map/dictionary for the existing rhymes?

answer = TO\_LOWERCASE(GET\_USER\_INPUT(rl))

**if** answer is equal to yes

PRINT rhymesData

PRINT “Do you want to exit the app?

answer = TO\_LOWERCASE(GET\_USER\_INPUT(rl))

**if** answer is yes

CLOSE\_APP()

**else**

START\_APP()

**END if**

**else**

CLOSE\_APP()

**END if**

**END if**

END FUNCTION START\_APP()

CLOSE\_APP()

PRINT “Exiting the app now…”

EXIT\_PROCESS()

START\_APP()

**Section 4:**

I have opted to utilize the Dynamic Array and Associative Array (Map) abstract data structures for their simplicity and efficiency in the algorithm. By avoiding the use of a dynamic array and comparing each input with the entire array for repeated word searches, we can conserve resources. Instead, we can perform this comparison only once for each user input word and define the corresponding rhymes, which are then assigned as key-value pairs in an associative array (map). This approach allows for efficient retrieval of values by key using the associative array for subsequent searches of the same word, as it operates in constant time complexity (O(1)), whereas searching an array would typically require linear time complexity (O(n)).

**Section 5:**

Link to approx. 1-minute YouTube video of the program and showing it running/working:

https://youtu.be/7PWDl4xmfHg

//ADS1 MidTerm Rhymes code implementation

var dataArray = [];

var rhymesData = new Map();

//loading the words file into an array

var fs = require('fs');

var wordListPath = './wordlist.csv';

var data = fs.readFileSync(wordListPath, 'utf-8');

dataArray = data.split('\n');

//creating the user interface

//This code for creating user interface was taken and adapted for my app from:

// https://www.codecademy.com/article/getting-user-input-in-node-js

const readline = require('readline');

    const rl = readline.createInterface({

        input: process.stdin,

        output: process.stdout

    });

//script logic

function startApp() {

    //this will compare all words and add the ones ending in same 3 letters to array

    function getRhyme(word) {

        var rhymesWords = [];

        //if they are part of the map/dictionary, we skip traversing the whole array,

        //else we continue with traversing

        if (rhymesData.has(word)) {

            rhymesWords = rhymesData.get(word);

        }

        else {

            for (var i = 0; i < dataArray.length; i++) {

                if (dataArray[i].slice(-4).trim() === word.slice(-3).trim()){

                    rhymesWords.push(dataArray[i]);

                }

            }

            //add the current word search to the map

            rhymesData.set(word, rhymesWords);

        }

        return rhymesWords;

    }

    //user input

    console.log('We have a total word count of ' + dataArray.length + ' in our database');

    rl.question('Enter a word to find a rhyme: ', (word) => {

        var rhymes = getRhyme(word);

        for (var i = 0; i < rhymes.length; i++) {

            console.log(rhymes[i]);

        }

        console.log('There are a total of ' + rhymes.length + ' rhymes for ' + word);

        //logic to restart the app or to show the map of key/value pairs for rhymes

        //already found/searched

        rl.question('Do you want to type another word? (yes/no) ', (answer) => {

            if (answer.toLowerCase() === 'yes') {

                startApp();

            }

            else {

                rl.question('Do you want to see the map/dictionary for the existing

rhymes? If you want to exit the app, type no! (yes/no) ', (answer) => {

                    //this will print the map to the console

                    if (answer.toLowerCase() === 'yes') {

                        console.log(rhymesData);

                        rl.question('Do you want to exit the app? (yes/no)', (answer) => {

                            if (answer.toLowerCase() === 'yes') {

                                closeApp();

                            }

                            else {

                                startApp();

                            }

                        })

                    }

                    else {

                        closeApp();

                    }

                })

            }

        })

    })

}

//closing the app

function closeApp() {

    console.log('Exiting the app now...');

    process.exit();

}

//initial call to start the app once the file is opened

startApp();

**Section 6:**

While my implementation successfully provides basic rhyme-finding functionality, it exhibits certain limitations that could be addressed for enhanced performance and user experience:

* **Rhyme Matching Algorithm**: The current approach of comparing the last three letters might yield inaccurate or incomplete results. A more comprehensive rhyming algorithm, such as considering phonetic patterns or using a dedicated rhyming dictionary, would significantly improve accuracy.
* **User Input Handling**: The repetitive questions for continuing or exiting the app could be streamlined. Implementing a menu-driven interface or using command-line arguments would provide a more user-friendly experience.
* **Error Handling**: The code doesn't explicitly handle potential errors, such as invalid user input or file access issues. Incorporating robust error-handling mechanisms would ensure graceful responses to unexpected situations.