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 efficiently find and retrieve rhymes, the app employs an algorithm that becomes more intelligent with each use. Initially, the CMU dictionary list of words is imported into a dynamic array. Subsequently, the words and their pronunciations are separated and stored as key-value pairs in an associative array (map), ensuring efficient storage and access.

When a user enters a word, a specialized rhyme-finding function is invoked. This function first checks if the user's word is already present in a 'rhyme dictionary'—a dedicated associative array (map) storing previously discovered rhymes. If the word is found, the corresponding word-rhyme pair is retrieved and displayed in the console.

If the word is not part of the 'rhyme dictionary,' the function proceeds to obtain the pronunciation of the word from the associative array containing all words and pronunciations. It then iterates over this array, calling another function to determine if the user's input pronunciation's phonetic representation is similar to any other words' pronunciations stored in the associative array. If a match is found, the word is added to a temporary dynamic array. After iterating over all the words from the associative array, having the temporary dynamic array filled with found words, the key-value pair is added to the 'rhyme dictionary' associative array for more efficient future retrieval of rhymes for the same word. Once completed, the dynamic array is displayed in the console.

Following a rhyme search, the user is prompted to continue typing another word, review previously searched rhymes, or exit the app entirely.

This optimization strategy excels during repeat searches. By pre-storing rhymes in the 'rhyme dictionary' associative array, unnecessary re-explorations of the associative array and phonetic representation similarities are avoided, ensuring swift rhyme delivery and a seamless, enjoyable user experience for frequent word searches. Retrieving a value by its corresponding key in the associative array is a fast operation that doesn't require looping through the entire array for repeated words. The algorithm gains speed with each use, as the 'rhyme dictionary' associative array grows with each new search.

Section 3:

```
wordMap = CREATE_ASSOCIATIVE_ARRAY()

existingRhymes = CREATE_ASSOCIATIVE_ARRAY()

lines = CREATE_DYNAMIC_ARRAY()

fileData = LOAD_FILE(WORD_LIST_PATH)

lines = SPLIT_LINES(fileData)
```

```
for i = 1 to LENGTH(lines)
        (word, pronunciation) = EXTRACT_WORD_AND_PRONUNCIATION(line)
        ADD(word, pronunciation, wordMap)
CREATE_READLINE_INTERFACE(rl)  //basic rl interface algorithm, credit given in the actual code implementation
FUNCTION START_APP()
        PRINT prompt providing the total number of words in database LENGTH(lines)
        PRINT "Enter a word to find a rhyme: "
                word = TO_LOWERCASE(GET_USER_INPUT(rl))
                PERFORM_RHYME_MATCHING(word)
                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 existingRhymes
                                                PRINT "Do you want to exit the app?
                                                        answer = TO_LOWERCASE(GET_USER_INPUT(rl))
                                                        if answer is yes
                                                                CLOSE_APP()
                                                        else
                                                                START_APP()
                                        else
                                                CLOSE_APP()
        FUNCTION PERFORM_RHYME_MATCHING(word)
                rhymes = CREATE_DYNAMIC_ARRAY()
                k = 1
                if HAS_KEY(word, existingRhymes)
                        rhymes = GET(word, existingRhymes)
                else
                        userWordPronunciation = GET(word, wordMap)
                        for each (mapWord, mapPronunciation) IN wordMap
```

```
                    if mapWord == word
                        CONTINUE
                    If PHONETIC_REPR_SIMILAR(userWordPronunciation, mapPronunciation)
                        rhymes[k] = mapWord
                SET(word, rhymes, existingRhymes)
            PRINT “Rhymes: “
            for i = 1 to LENGTH(rhymes)
                PRINT rhymes[i]
            PRINT “Total number of rhymes LENGTH(rhymes)”
        END FUNCTION PERFORM_RHYME_MATCHING()
        FUNCTION PHONETIC_REPR_SIMILAR(pronunciation1, pronunciation2)
            distance = LEVENSHTEIN_DISTANCE(pronunciation1, pronunciation2)
            similarityThreshold = 2
            return distance <= similarityThreshold
        END FUNCTION PHONETIC_REPR_SIMILAR()
        FUNCTION LEVENSHTEIN_DISTANCE(word1, word2)
        //basic Levenshtein distance algorithm, credit given in the actual code implementation
        END FUNCTION LEVENSHTEIN_DISTANCE()
END FUNCTION START_APP()
CLOSE_APP()
        PRINT “Exiting the app now…”
        EXIT_PROCESS()
START_APP()
```

Section 4:

I chose to implement 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 eliminating the need to compare each input with the entire array during repeated word searches, we can optimize resource utilization. Instead, the algorithm performs the phonetic comparison only once for each user-input word and establishes corresponding rhymes. These rhymes are then stored as key-value pairs in an associative array (map). This approach facilitates efficient retrieval of values by key using the associative array for subsequent searches of the same word, operating with constant time complexity (O(1)). In contrast, searching an array would typically demand linear time complexity (O(n)). The adoption of these data structures contributes to the algorithm's speed and resource efficiency, particularly beneficial during frequent word searches.

Section 5:

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

https://youtu.be/7pHzE3dKXv4

```
//ADS1 MidTerm Rhymes App
//loading the CMU dictionary version 0.7b
const fs = require('fs');
const wordListPath = './wordlist.csv';
const wordMap = new Map();
const existingRhymes = new Map();
const fileData = fs.readFileSync(wordListPath, 'utf-8');
const lines = fileData.split('\n');


//this will separate the words from their pronunciation
//as soon as first space in the word is found
for (let i = 0; i < lines.length; i++) {
    const line = lines[i].trim();
    if (line !== '') {
        const firstSpaceIndex = line.indexOf(' ');
        //extract the word from the line
        const word = line.slice(0, firstSpaceIndex).trim();
        //extract the pronunciation from the line, trimming any additional spaces
        const pronunciation = line.slice(firstSpaceIndex + 1).trim();
        wordMap.set(word.toLowerCase(), pronunciation);
    }
}
//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
});

function startApp() {
    //user input
    console.log('There are a total of '+wordMap.size+' words in the database')
    rl.question('Enter a word to find rhymes: ', (userInput) => {
        const wordToFindRhymesFor = userInput.trim().toLowerCase();
        performRhymeMatching(wordToFindRhymesFor);
        rl.question('Do you want to type another word? ', (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(existingRhymes);
                        rl.question('Do you want to exit the app? (yes/no) ',(answer) => {
                            if (answer.toLowerCase() === 'yes') {
                                closeApp();
                            }
```

```
                        else {
                            startApp();
                        }
                    })
                }
                else {
                    closeApp();
                }
            })
        }
    })
});


//function to perform rhyme matching for the user input word
//catching any new rhyme searches in a new map for easy future retrieval
function performRhymeMatching(word) {
    let rhymes = [];
    if(!existingRhymes.has(word)) {
        const userInputPronunciation = wordMap.get(word);
        //iterating over the words in the wordMap
        for (const [mapWord, mapPronunciation] of wordMap) {
            if (mapWord === word) {
                continue;
            }
            //comparing the phonetic representations
            if (arePhoneticRepresentationsSimilar(userInputPronunciation,
                mapPronunciation)) {
                rhymes.push(mapWord);
            }
        }
        existingRhymes.set(word, rhymes);
    }
    else {
        rhymes = existingRhymes.get(word);
    }
    //displaying the rhymes
    console.log('Rhymes: ');
    for (var i = 0; i < rhymes.length; i++) {
        console.log(rhymes[i]);
    }
    console.log('There are a total of ' + rhymes.length + ' rhymes for ' + word);
}

//function to compare the phonetic representations and
//determine if they are similar with similarity threshold
function arePhoneticRepresentationsSimilar(pronunciation1, pronunciation2) {
    const distance = levenshteinDistance(pronunciation1, pronunciation2);
    const similarityThreshold = 1.5;
    return distance <= similarityThreshold;
}
//the levenshtein distance algorithm was taken and adapted from
//https://www.tutorialspoint.com/levenshtein-distance-in-javascript
function levenshteinDistance(word1, word2) {
```

```
        const m = word1.length;
        const n = word2.length;
        const dp = Array.from(Array(m + 1), () => Array(n + 1).fill(0));
        for (let i = 0; i <= m; i++) {
            dp[i][0] = i;
        }
        for (let j = 0; j <= n; j++) {
            dp[0][j] = j;
        }
        for (let i = 1; i <= m; i++) {
            for (let j = 1; j <= n; j++) {
                if (word1[i - 1] === word2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1];
                }
                else {
                    dp[i][j] = Math.min(
                        dp[i - 1][j - 1] + 1,
                        dp[i][j - 1] + 1,
                        dp[i - 1][j] + 1
                    );
                }
            }
        }
        return dp[m][n];
    }
}
//closing the app
function closeApp() {
    console.log('Exiting the app now...');
    process.exit();
}

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 phonetic similarities using the CMU dictionary might yield inaccurate or incomplete results. A more comprehensive rhyming algorithm, such as utilizing 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.
- **Exploring advanced rhyming techniques**: The implementation of a soundex function or leveraging machine learning models, could further refine the rhyme-matching capabilities. These advanced techniques have the potential to identify subtle phonetic similarities, providing more accurate and nuanced rhyme suggestions.