TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

PROJECT REPORT

TOPIC

DICTIONARY

Theoretical lecturer: Đinh Bá Tiến

Teacher Assistant: Hồ Tuấn Thanh
Trương Phước Lộc

HCM, 21/08/2023

Mục lục

1 GROUP INFORMATION

Members	ID	Contribution Percentage
Lương Nguyên Khoa	22125040	25%
Nguyên Võ Hoàng Thông	22125103	25%
Lê Phát Minh	22125056	25%
Nguyễn Hữu Quốc Thắng	22125091	25%

2 INTRODUCTION

"The Dictionary" is a project within the APCS program at the University of Science, HCMC (HCMUS). Its aim is to showcase our understanding of various data structures and algorithms and how they can solve real-world problems. While the program leans towards a prototype rather than a professional solution, it offers complete functionality and a good user experience.
This report outlines the data structures and algorithms used, as well as the reasons behind our design choices. We'll dive into technical solutions, but won't provide in-depth explanations of data structures, as they aren't the primary focus here. Additionally, we'll include a manual with illustrations on how to use each function.

3 DATA STORAGE

The program's permanent data is stored in data/, including 5 text file datasets of 5 types of dictionary: eng-eng, eng-vie, vie-eng, emoji, slang. When the program runs, it will generate temporary data in tmp/, divided into 5 subfolders for 5 datasets, each subfolder contains:

- data.dict: data serialized from trie data structure, which will be discussed later.
- history.txt: store history of word searched.
- favorite.txt: store user favorite entries.

4 PROBLEM AND SOLUTIONS

A dictionary application requires good data management and refined usage of algorithms. First, every word has its own content, formatted differently depending on the dataset. Therefore, a Class representing a word is necessary, and how their fields (information stored) are organized so that a single class can fit all 5 datasets is a considerable issue. Insights on how we handle this problem is in section 4.
Another problem is that we cannot use naive data structures, such as arrays for storing and searching words. Searching is the primary operation used on dictionary apps and therefore needs to run in an instant. Linear or even binary search algorithms are not good enough when it comes to searching datasets of more than 100000 entries, which our program uses. Array is also not a good data structure for various features needed by a dictionary, search as word suggestion that is updated per character input. To resolve this problem, our group decided to use the trie data structure as the primary data structure for this project.

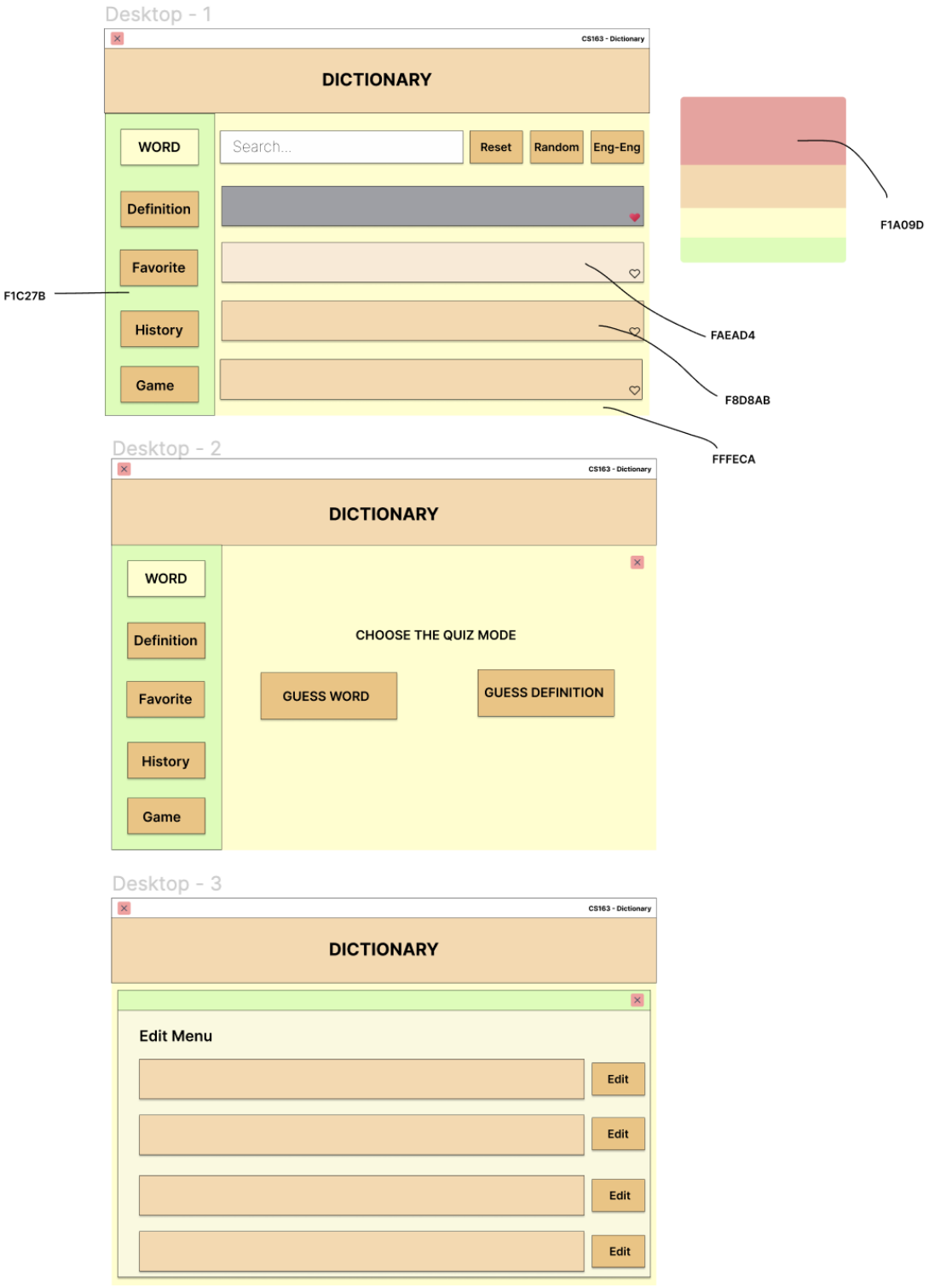
Lastly, the required feature for searching from definition is an issue that although trie is still somewhat appropriate, it compromises a lot of criteria such as memory usage; the height of the trie will also be significant, which reduces its performance. Hash table is what we choose as an attempt to tackle the aforementioned problem, because it gains us the flexibility in implementing behaviors that we would like our product to have (key word of a definition will be displayed if the definition reaches a threshold of similarity).

5 PROGRAM STRUCTURE

This section is divided into two parts: the backend and the frontend. In the backend section, we will discuss how our Classes and their member functions work and how they solve problems that naive approaches may not. The frontend section will showcase the initial and final designs, without delving deeply into algorithms and code. Our primary focus has been on the backend to ensure a robust and efficient system. Since the program doesn't heavily rely on data storage methods, unlike other programs such as course management software, we will first list the source files, the Classes they contain, and then explain their member functions and behaviours.

5.1 Frontend

We utilize the Raylib and Raygui libraries to code the interface of this program. It includes all the source file (with .cpp extension) and their header (.h extension) that are in include/frontend/ (for header files) and src/fronted (for source files).
Initially, we sketched a blueprint using Figma and used it as a reference to implement the complete GUI. However, during the development process, we made several design changes, and this is the final outcome of our efforts.



Hình 1: Initial design

Hình 2: Final design

5.2 Backend

This is the core of the program which provides functionality. It includes all the source files (with .cpp extension) and their header (.h extension) that are in include/dictionary/ (for header files) and src/-dictionary/ (for source files). Specifically, they are build.cpp (and build.h), trie.cpp (and trie.h), random.cpp (and random.h), history.cpp (and history.h), searchDefinition.cpp (and searchDefinition.h), word.cpp (and word.h).

5.2.1 Word

Class Word:
The Word class represents a word in a dictionary and contains information such as the word's key, type, and one or more definitions. This class contains:

Private field:

- A string to store the keyword of a word.
- A string to store the type of the word.

- A vector of strings to store the definitions of a word.
- Auxiliary functions.

Public field:

- Auxiliary functions help interact with the private variables instead of handling it externally which may lead to undesired behaviors. As the function identifiers themselves are self-explanatory, there are no other means to better explain them than the source code:

```
class Word
{
  private:
    std::string key;
    std::string type;
    std::vector<std::string> definitions;

  public:
    Word();
    Word(std::string key, std::string type = "", std::string definition
    = "");
    std::string getKey();
    std::string getType();
    std::string getDefinition(int index);
    std::vector<std::string> getDefinitions();
    int getDefinitionCount();
    void setKey(std::string key);
    void setType(std::string type);
    void setDefinition(std::string definition, int index);
    void addDefinition(std::string definition);
    void removeDefinition(int index);
    void editDefinition(int index, std::string edit);
};
```

5.2.2 Class Dictionary

The datasets used for the program are plain texts with predefined syntaxes for storing keywords and their definitions. The Dictionary class is a representation of the datasets in the program (but not the main data structure used for core dictionary-related features). It allows the callee program to have a simple interface for extracting data from datasets without having to consider the complexity of how each type of dataset are processed (there are 5 types of dataset namely eng-eng, eng-vie, vie-eng, emoji, and slang).

Private field:

- A std::ifstream object from stream to read the data from the file.
- An integer number denoting the type of dictionary being used.
- 5 functions to extract each word from the datasets, each function is written for its corresponding type of dataset.

Public field:

- Constructor where file path and dictionary type must be specified.
- A wrapper function to extract a word from the dataset upon calling. It will call one of the 5 word-extracting functions from the private field based on the dictionary type given.
- Other auxiliary functions to check for the end of the file or get the dictionary type.

The header is as below:

```
class Dictionary
{
  private:
    std::ifstream fin;
    int dictType; // 0: eng-eng, 1: eng-vie, 2: vie-eng, 3: emoji, 4:
    slang
    Word getWordEngEng();
    Word getWordEngVie();
    Word getWordVieEng();
    Word getWordSlang();
    Word getWordEmoji();

  public:
    Dictionary(std::string path, int dictType);
    ~Dictionary();
    Word getWord();
    int getDictionaryType();
    bool eof();
};
```

5.2.3 Class Trie

The Trie class is an implementation of the trie data structure using the aforementioned TrieNode struct as its nodes. This is the core data structure for dictionary-related features as it is highly efficient in functionalities such as searching or removing words, as well as word suggestion given a word's suffix. It utilizes a tree-like structure where each node represents a character in a word. More detail on how a trie works, in general, is given in Reference, as this section will only focus on the authors' implementation of trie to fit the requirements of the program. Here is a quick look at the Trie object structure:

- A pointer to the root of the trie.
- Functions to insert, delete and search for a word.
- Functions to suggest words.
- Other auxiliary functions.

Here is the prototype of the object:

```
const int ALPHABET = 128;
struct TrieNode
{
    TrieNode *children[ALPHABET];
    bool endOfWord;
    Word word;
};

class Trie
{
  private:
    TrieNode *root = nullptr;
    TrieNode *createNode();
    TrieNode *remove(TrieNode *root, std::string key, int index);
    void clear(TrieNode *root);
    bool isEmptyNode(TrieNode *node);
    void serialize(TrieNode *root, std::ofstream &fout, char delimiter)
    ;
    void deserialize(std::ifstream &fin, char delimiter);
    void wordSuggest(std::vector<Word> &wordlist, int limit, std::queue
    <TrieNode *> &q);

  public:
    Trie();
    ~Trie();
    void insert(Word word);
    // bool prefixSearch(std::string key);
    bool search(std::string key, Word &word);
    void remove(std::string key);
    std::vector<Word> wordSuggest(std::string prefix);
    void serialize(std::string path, char delimiter);
    void deserialize(std::string path, char delimiter);
    void clear();
};
```

5.2.4 Detailed explanation

The **TrieNode** class is a struct that contains the following fields:

- **children:** an array of pointers to its children nodes. The size of the array is equal to the number of characters used. The program uses 128 characters (ASCII) as the trie is also used for the emoji dataset which contains symbols outside of the alphabet.
- **endOfWord:** a boolean flag that indicates whether the node represents the end of a word, and therefore forms a complete word.
- **word:** an object of type Word.

The **Trie** class is the main class that implements the Trie data structure. It has the following member functions:

- **void insert(Word word):** inserts a word into the Trie. Time complexity: O(N), N is the length of the keyword of the Word object.

- **bool search(std::string key, Word word):** searches for a word in the Trie and returns true or false based on whether the key is found or not. If the keyword is found, return the Word object representing that word's content by reference through the parameter. Time complexity: O(n), n is the length of the keyword of the Word object.
- **void remove(std::string key):** removes a word from the Trie. Time complexity: O(N), N is the length of the keyword of the Word object.
- **std::vector<Word> wordSuggest(std::string prefix):** returns a vector of Word objects that match a given prefix. It has a limit of 15 words intentionally set (because finding more is unnecessary). The function uses Breadth First Search (BFS) to traverse and retrieve words with similar prefixes instead of Depth First Search (DFS) to comply with the common dictionary's word order rule. Therefore, the time complexity is O(n), with n being.
- **void serialize(std::string path, char delimiter):** serializes the Trie to the data.dict file stored in tmp/. This function will be called by the build function in 2.1.3. upon the first launch of the program in order to store the trie built from the raw dataset to the data.dict, which is basically a processed version of the raw dataset that can be retrieved faster into memory (this function is made specifically for performance improvement). The delimiter parameter decides how different information of the Word object is divided in the file so that it can be re-read easily. Time complexity: O(n), n is the number of words being serialized, or in other words the number of entries in each dictionary dataset.
- **void deserialize(std::string path, char delimiter):** deserializes data stored in tmp/data.dict to the Trie. This function will be called by the build function in 2.1.3 from the second launch of the program onwards. This speeds up the startup time of the program by a good margin (50 from the authors' observation), compared to building from raw datasets every time the program is launched. Time complexity: O(n), n is the number of words being deserialized, or in other words the number of entries in each dictionary dataset.
- **void clear():** clears the whole Trie from memory. Time complexity: O(N), N is the number of nodes in the trie. This operation is performed every time the program exits.

The **Trie** class also has several private member functions that act as the helper functions to their counterparts in the public field:

- **TrieNode* createNode():** return a pointer to a new TrieNode, with all null pointers as children, false value in the endOfWord variable and an empty Word object.
- remove(TrieNode *root, std::string key, int index): for recursion.
- **clear(TrieNode *root):** clears all nodes of the trie starting from a particular node. By default, the clear() function will pass the root of the Trie to this helper function.
- **isEmptyNode(TrieNode *node):** checks if a node is empty, used internally by other functions.
- **serialize(TrieNode *root, std::ofstream fout, char delimiter):** serialize a node to a file from a given node as root, an ofstream object corresponding to a dataset path and a delimiter given by the caller function. By default, the serialize function from the public will pass the root of the trie as a parameter to this function.

- **deserialize(std::ifstream fin, char delimiter):** helper for deserialize function.
- **wordSuggest(std::vector<Word> wordlist, int limit, std::queue<TrieNode*> q):** helper function of wordSuggest in the public field, used for performing BFS.

Overall, our trie is implemented in a fairly common way, with the exception of the wordSuggest function and an additional Word object stored in each node, for ease of retrieving a word's content.

5.2.5 SearchDefinition

Class Relevant Word:
This class represents a word with its relevance to the input of the users. Words with high relevance to the user's input will be shown. In detail, this class contains:

Private field:

- A pointer to the Dictionary object representing the dictionary being used.
- A string containing the path to the dataset.
- An object Word to store the word and its context (its keyword, its type, and definitions).
- A double number to show the relevance of a word to the given context.

Public field:

- **std::vector<RelevantWord> searchDefinition(std::string definitionFromUser, Trie trie):** this function performs a search based on the user-provided definition and returns the vector of words that are relevant to the context. (Detailed algorithm about this function will be explained in the next section).

+ Helper function:

- **void wordHashing (std::string line, std::unordered_set<std::string> & wordCounts):** This function hashes all words appearing in the content of a word (specifically its definitions).
- **double calculateRelevance(std::string userInput, std::unordered_set<std::string>& wordCounts):** This function calculates the relevance of a word compared to the user-provided definition.

Here is the overview of the RelevantWord Class:

```
class RelevantWord
{
  private:
    Dictionary *dictionary;
    std::string path;
    Word word;
    double relevance;

  public:
```

```
    RelevantWord();
    RelevantWord(Word word, double relevance);
    void setDictionary(Dictionary *dict);
    void setPath();
    std::string getPath();
    std::vector<RelevantWord> searchDefinition(std::string
   definitionFromUser, Trie &trie);
    double getRelevance();
    Word getWord();
};

// support functions
bool comparingRelevance(RelevantWord a, RelevantWord b);
std::string preprocessText(std::string text);
void wordHashing(std::string line, std::unordered_set<std::string> &
   wordCounts);
double calculateRelevance(std::string userInput, std::unordered_set<std
   ::string> &wordCounts);
```

5.2.6 Random

Class Random:
The **Random** class represents a component that generates random word-related tasks or quizzes based on a given dictionary. This class includes:

Private field:

- A pointer to the Dictionary object representing the dictionary being used.
- A pointer to the Dictionary object representing the dictionary being used.
- A string containing the path to the dataset.

Public field:

- **Word viewRandomWord():** this function generates a random word in the dataset and returns an object Word (containing its keyword, type, and definitions). The code iterates through the file to find the exact keyword and return that word, therefore, the time complexity of this code is O(n) with n as the number of lines that the code reads.
- **std::string guessDefinition():** this function randomly chooses a keyword and its definition, then it randomly chooses 3 more definitions from the dataset to form a quiz. The algorithm is just the same as that of viewing a random word. The time complexity of this code is O(n) with n as the number of lines that the code reads.
- **std::string guessKeyWord():** just like the function guessDefinition above, this function chooses a keyword and its definition and then it gets 3 more keywords to form a quiz. The algorithm is the same as that of viewing a random word, therefore, the time complexity is also O(n) with n the number of words that the code reads.

- Other auxiliary functions.

Here is the overview of the Random Class:

```
class Random
{
  private:
    Dictionary *dictionary = nullptr; // dictionary
    std::string path;                 // path to dataset
  public:
    void setDictionary(Dictionary *dict);
    void setPath();
    std::string getPath();
    // Helper function
    int getRandomNumber();
    int randomFourAnswer();
    int randomChoice = 0;
    int getChoice();
    Word viewRandomWord(); // Function for viewing a random word

    // The structure of the vector<string>
    // index 0: question (either a key word or a definition)
    // index 1: correct answer (either a key word or a definition)
    // index 2-3-4: wrong answer (either a key word or a definition)

    std::vector<std::string> guessDefinition(); // Function returns a
    vector<string> containing
                                                // 4 definitions and 1
    key word

    std::vector<std::string> guessKeyWord(); // Function returns a
    vector<string> containing
                                            // 4 key words and 1
    definition

    // void quizWith4Definitions();
    // void quizWith4KeyWords();
};

// Quiz
std::string get1stDefinitionFromText(const std::string &line);
std::string getKeyWordFromText(const std::string &line);
```

5.2.7 History

Class History:
The **History** class represents a history tracker that stores and manages a collection of strings. It provides functionality to add, remove, find, retrieve, and save strings to a storage system, typically backed by a file. Generally, this class loads the history.txt file into the memory as a vector of strings of words in history, add an entry with every word searched and write to the file when the program terminates. It can also be used for the favorite tracker as favorite is just a conditioned type of history. This class consists of:

Private field:

- An integer number denoting the type of the history, 0 for history and 1 for favorite. This will determine whether the list of words is sorted in alphabetical order (for favorite entries) or not.
- A string storing the path to the history file of that dataset. This is need for loading entries from text file to memory when the object is created and writing back into the text file when the object is destroyed.
- A vector of strings to store words in history/favorite.

Public field:

- **Constructor:** construct a History object with a specified file path and a mode.
- **int find(std::string key):** Searches for a specified key in the history data and returns the index of the first occurrence if found, otherwise returns -1. Time complexity: O(n), n is the number of word entries.
- **void add(std::string key):** Adds a new key to the history data in memory without modifying the file. If not, add it to the internal vector. Otherwise, does nothing if the class is in favorite mode or remove the old entry and push the new entry at the end of the vector. Time complexity: O(n), n is the number of word entries as it has to call the find function to check whether the word exists.
- **void remove(std::string key):** Removes a specified key from the history data in memory, if the word exists. Time complexity: O(n), n is the number of words as it has to call the find function to find the index of the to-be-removed word.
- **void save():** Modifies the file by persisting the current history data from memory into the file. Time complexity: O(n), n is the number of entries.

Here is the overview of the History class:

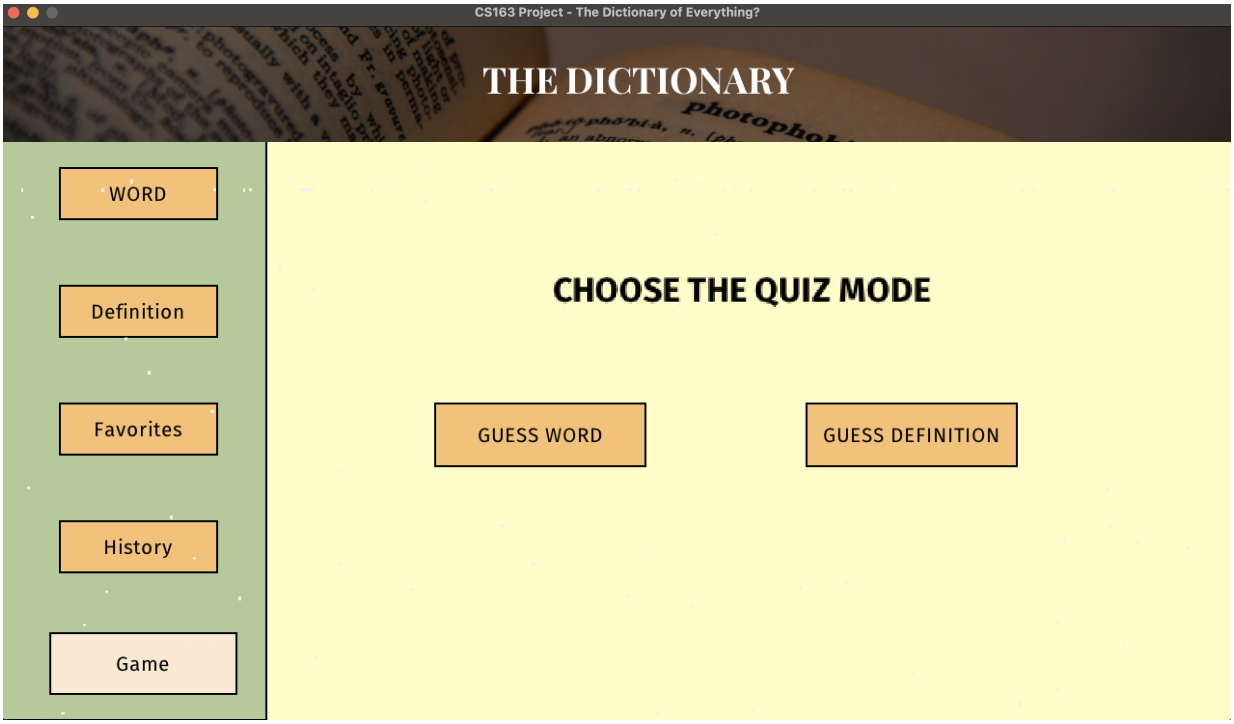
```
class History
{
  private:
    std::string path;
    int mode;
    std::vector<std::string> storage;

  public:
    History(std::string path, bool mode = 0);
    ~History();
    int find(std::string key);
    std::vector<std::string> get();
    void add(std::string key);    // add to memory but does not modify
   the file
    void remove(std::string key); // remove from memory but does not
   modify the file
    void save();                  // modify the file
    void clear();                 // clear the memory
};
```

6 DESIGN AND PERFORMANCE

6.1 Design Overview

Hình 3: Landing Page

1: Search by using the keyword.
2: Search by using the definition.
3: View your favorite words.
4: View your previous search words.
5: Game mode. You can choose either guess a keyword or guess a definition. (Picture 2).
6: Reset the dictionary (returning to the original version).
7: Random a word with its definitions.
8: Switch to other datasets. You can choose 5 different types: English-English, English-Vietnamese, Vietnamese-English, Emoji dictionary, and Slang dictionary.



Hình 4: Game Mode

Hình 5: Switch Dictionary

6.2 Loading Datasets

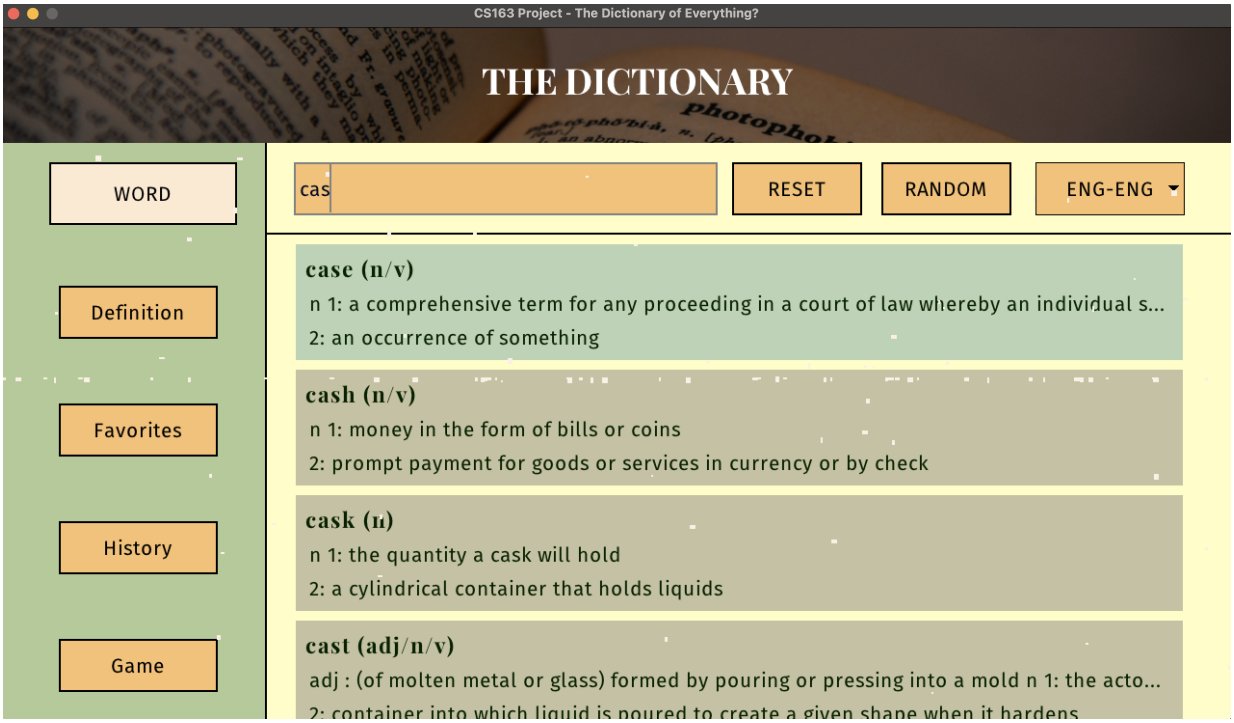
- At first, all 5 datasets are loaded into the system.

- A separate trie is created for every dataset, and it can be switched whenever users choose a different mode.
- The time complexity for this operation is O(n), where n is the number of words in each dataset.
- The space complexity is also O(n), where n is the number of words in each dataset.
- The total running time for loading all 5 datasets is approximately 3s.

6.3 Searching

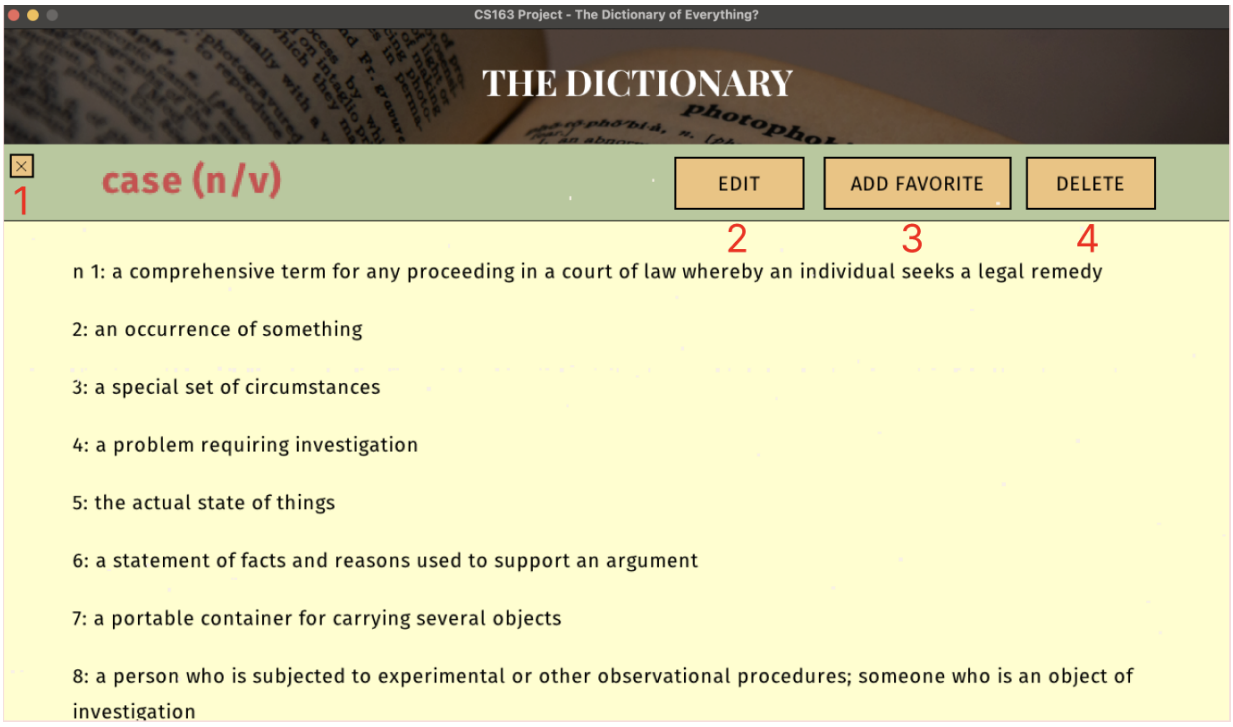
6.3.1 Search by keyword

This is the search screen for our product. The type of dictionary is English-English by default, and users can switch to other modes such as English-Vietnamese, Vietnamese-English, emoji dictionary, or slang dictionary.
While typing a keyword in the search bar, the dictionary will suggest some suitable words and the user can click on one of the options shown on the screen to view the keyword and its definitions.



Hình 6: Search Keyword

Users can choose the word:



Hình 7: Single Word Information

1: Close the window and back to the landing page.
2: Edit button to edit the definition of the word.
3: Add favourite and remove favourite.
4: Delete the word from the dictionary.

6.4 Algorithms

6.4.1 Searching

The system iterates through the string given by users and traverses through the trie. At the end of the string, if it's the end of a word, the system will show the word, and if not, there will be an option in which users can add this word to the dictionary (this feature will be mentioned later in this section).

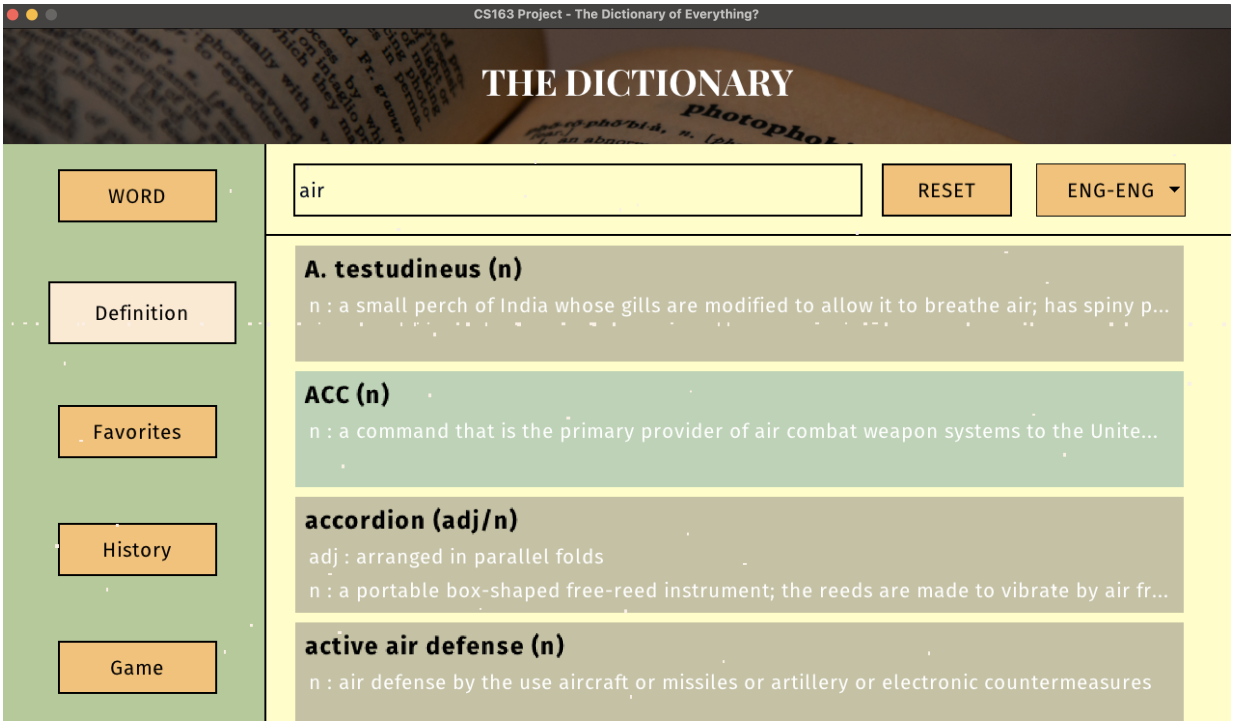
- The time complexity for this operation is O(m) with m is the length of the input.
- The running time: almost 0 seconds.

6.4.2 Word suggestion

- The code traverses the Trie based on the character in the prefix that users enter. If the prefix is not found in the Trie, an empty wordlist is returned.
- Once the Trie node corresponding to the prefix is reached, the code adds the word associated with that node to the wordlist if it represents the end of a word. It then performs a BFS traversal to find more suggestions by exploring the Trie nodes.

- The overall time complexity of the operation above is: O(k*V), k is the number of characters in the prefix, as each time the user input a new character into the existing string, the wordSuggest function is performed again; V is the number of vertices (nodes) in the trie. (more detailed expression on the previous section).
- Running time: almost in an instant (smaller than 0.5s).

6.4.3 Search by definition



Hình 8: Search by Definition

Searching by definition is just the same as searching by keyword. Users enter one or more keywords into the search bar and then the system will show words in alphabetical order that have at least 30 percent of relevance to the user's context.

6.4.4 Algorithm

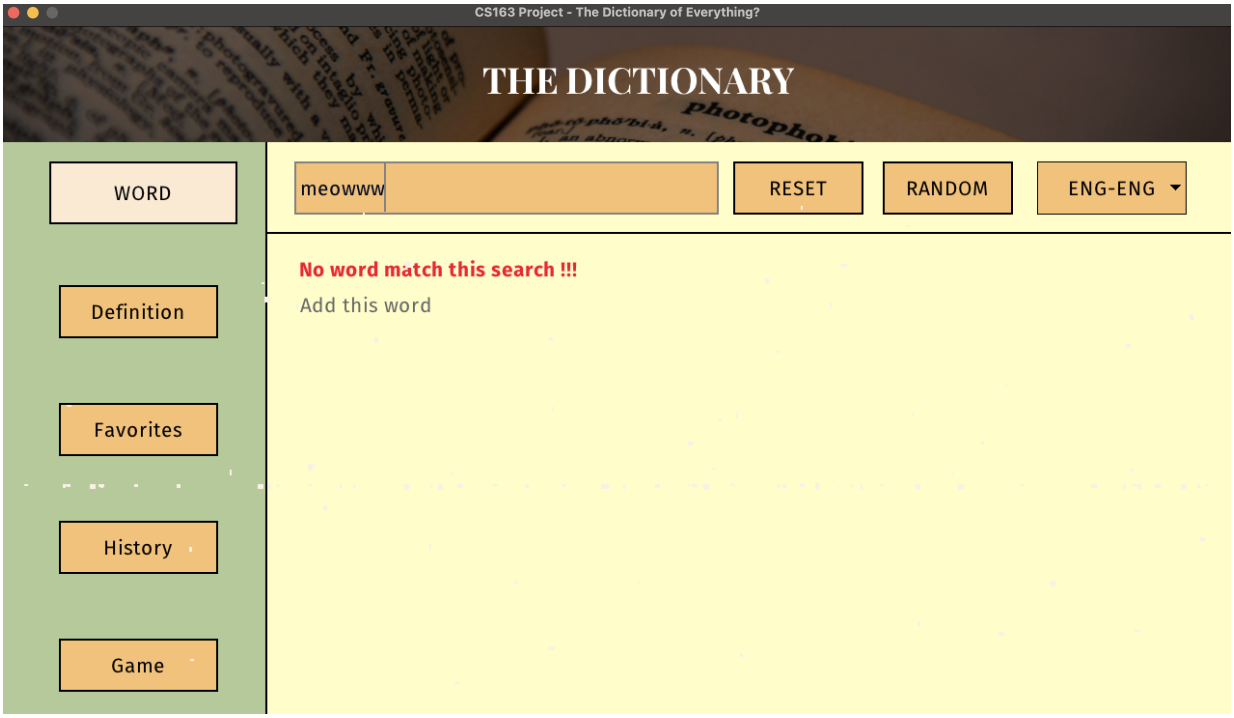
It transforms the definition to lowercase and opens input file streams for keyword and processed definition files. The code iterates through the files, calculating the relevance of the definition with each processed definition. If the relevance is above a threshold, an object RelevantWord will be created and stored in the vector.

- Time complexity: O(n*m) on average, with N as the number of words being read and n is the number of words on a line.
- Running time: approximately 0.3-0.8s on average, in the worst case - if the code reads the whole file, it can take up to 2s.

With the current limitations in our algorithmic knowledge, we have put forth our best efforts to create the most optimized algorithm. I believe that in the future, with the opportunity to learn more algorithms, we may be able to further optimize this search algorithm.

6.4.5 Adding a word

If users search by using a keyword but it doesn't exist in the dictionary, they can add it to the dictionary by clicking "Add this word".



Hình 9: Add new word

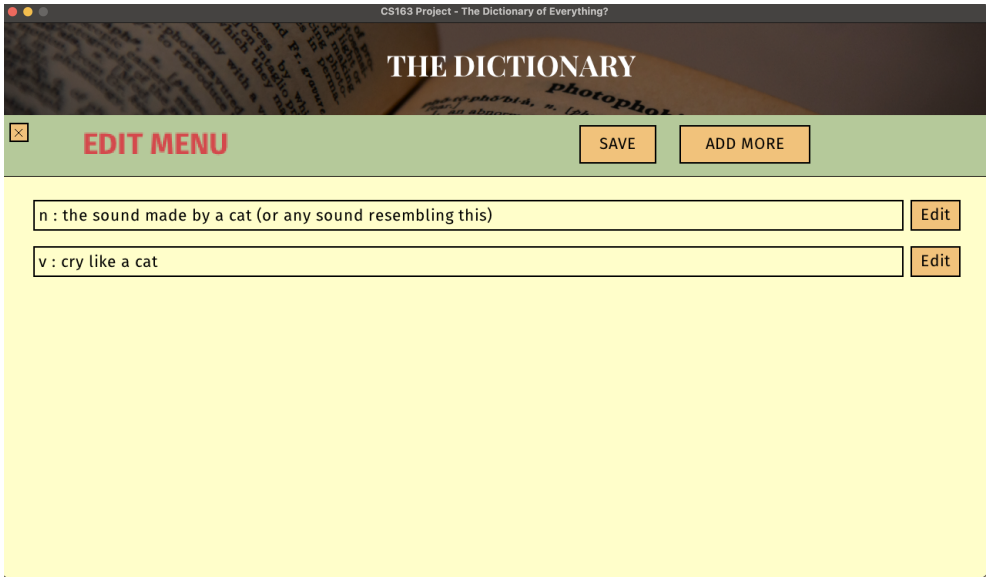
6.4.6 Algorithms

Create an object Word based on the user-provided input Insert it into the Trie.

- Time complexity: O(m) with m is the length of the keyword.
- Running time: very small.

6.4.7 Updating a word

In the list of definitions shown on the screen, you can click on the "Edit" button to edit one of the existing definitions.



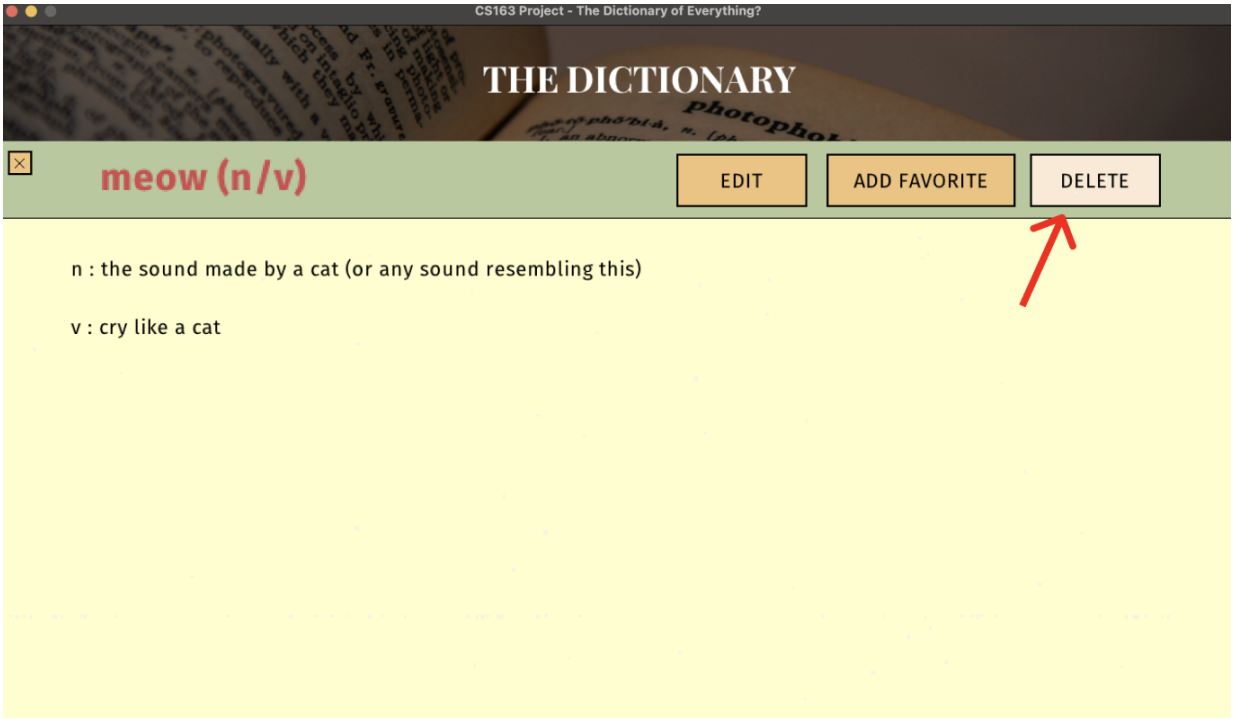
Hình 10: Edit Menu

After editing, users can click “Save” to save changes that they have made or click “Add More” to add more definitions. Algorithm: When users click “Edit” to one of the definitions, an index will return to the system, then the code gets the new definition that users enter, and then the system applies the new change.

- Time complexity: O(n), n is the number of definitions the word have, as the program has to check whether the definition is repeated or not.
- Running time: in an instant.

6.4.8 Delete a word

In the definition menu, users can also delete a word by clicking the “Delete” button.



Hình 11: Delete a Word

6.4.9 Algorithmss

- The system starts by checking if the root node is null, in which case it returns null. If the index reaches the end of the key, it marks the current node as not the end of a word. If the removed key results in an empty node, it is deleted and set to null.
- The function then proceeds to remove the key from the child node recursively. Finally, if the current node is both an empty node and not the end of a word, it is deleted and set to null.
- Time complexity: O(n) with n is the length of the keyword.
- Running time: very small.

7 REFERENCE

https://www.geeksforgeeks.org/trie-insert-and-search/

https://www.linkedin.com/advice/0/what-advantages-disadvantages-using-edit-distance#:~:text=The%20cosine%20similarity%20ranges%20from,document%20clustering%2C%20and%20information%20retrieval.