VIETNAM NATIONAL UNIVERSITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

ADVANCED PROGRAM IN COMPUTER SCIENCE

TECHNICAL REPORT

Project: A MINI DICTIONARY

Subject: CS163 - Data Structures

GROUP 4 - 23APCS2

Group members:
20125088 - Le Quoc Cuong
23125002 - Vo Minh Dang
23125007 - Nguyen Gia Khanh
23125030 - Phan Tan Dat
23125034 - Cao Thanh Hieu

Lecturers:
Ho Tuan Thanh, M.Sc.
Truong Phuoc Loc, M.Sc

August 30, 2024

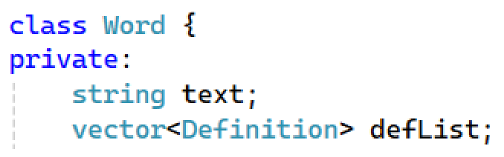
Contents

1 Base Classes and Searching Tools

We first go through base classes to understand the core of our program, then move on to 2 searching tools and evaluate their **algorithm, main functions, complexity and runtime**.

1.1 Word

Class Word represents the word object in our dictionary, its structure is shown below:



```
class Word {
private:
    string text;
    vector<Definition> defList;
```

Private:

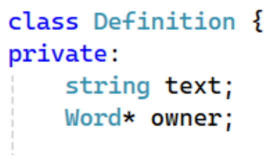
- **text:** characters of the word stored inside a string.
- **defList:** a vector of **Definition**. This class will be mentioned later on.

Public:

- auxiliary functions (getters, setters,...).

1.2 Definition

Class Definition represents a definition of a word. The word-definition relationship is one to many.



```
class Definition {
private:
    string text;
    Word* owner;
```

Private:

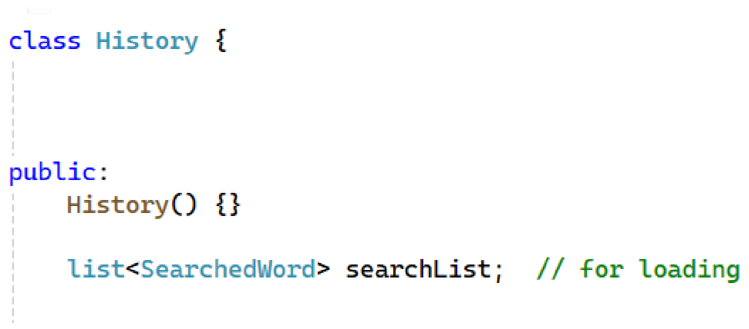
- **text:** characters of the definition stored inside a string.
- **owner:** A pointer to **Word** object that holds this definition.

Public:

- auxiliary functions (getters, setters,...).

1.3 History

Class History is used to track the searching history of user.



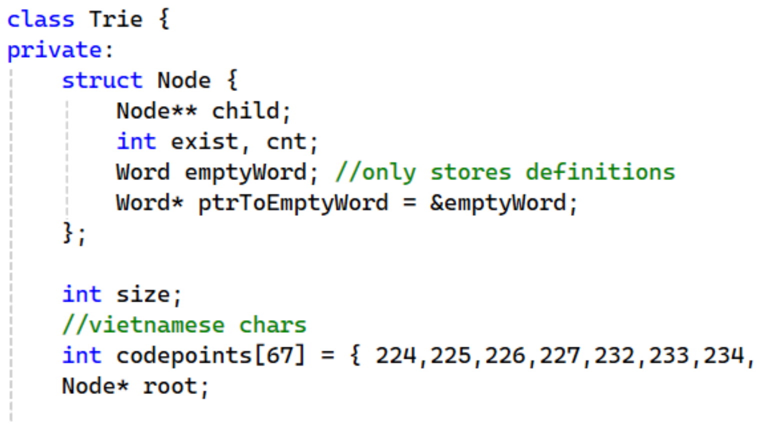
```
class History {



public:
    History() {}

    list<SearchedWord> searchList;  // for loading

```

Public:

- **searchList:** a list of searched words with respect to some dataset.
- auxiliary functions (getters, setters,...).

1.4 Trie

Trie is one of our reliable searching tools, designed specifically for searching word to definition. It can flexibly handle both Vietnamese and English words.



```
class Trie {
private:
    struct Node {
        Node** child;
        int exist, cnt;
        Word emptyWord; //only stores definitions
        Word* ptrToEmptyWord = &emptyWord;
    };

    int size;
    //vietnamese chars
    int codepoints[67] = { 224,225,226,227,232,233,234,
    Node* root;

```

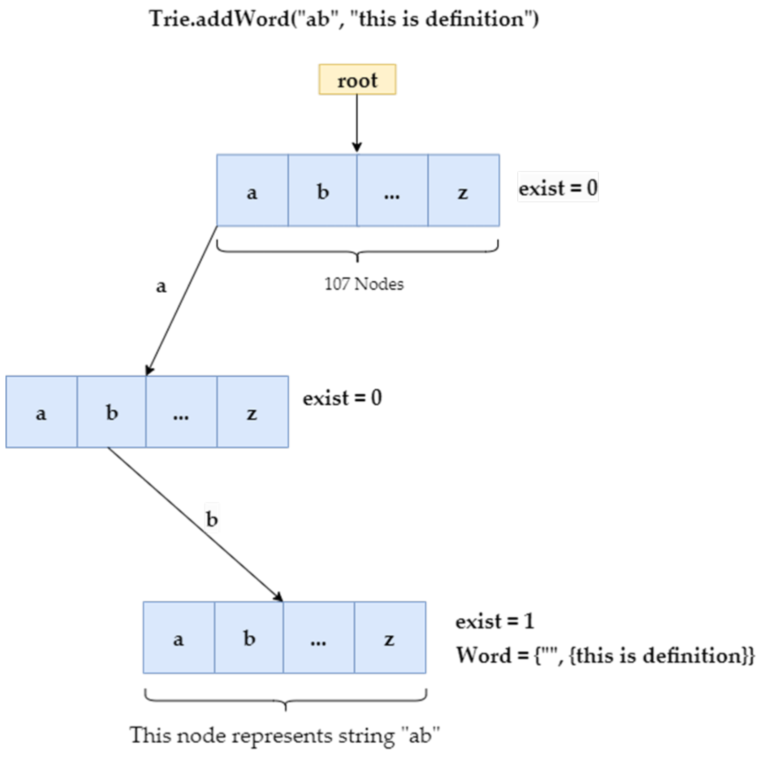
Private:

- **size**: an integer represents the number of nodes in trie.
- **codepoints[67]**: unicode values of Vietnamese characters.
- **root**: root pointer of our whole tree.

struct Node:

- **child**: an array of pointers to other nodes. In this case, **107** pointers to be precise.

- **exist**: an integer representing the existence of a string in our Trie (will be further explained below).
- **cnt**: counts how many times a specific character at a specific position has been added.
- **emptyWord**: a word that stores definitions of a string.
- **ptrToEmptyWord**: pointer to that word.


Trie.addWord("ab", "this is definition")
root
a
b
...
z
exist = 0
107 Nodes
a
exist = 0
b
exist = 1
Word = {"", {this is definition}}
This node represents string "ab"

Operating Method:

Trie is technically a *K-nary* search tree where in this case, $K = 107$, representing 107 characters that can appear in our word (many more than the alphabet of course).

To "store" a **Word** object with **text** $= s[1]s[2]...s[n]$ in this Trie, we will traverse the tree from root correspondingly to those characters (take a look at the picture above).

Formally, a word with structure mentioned above is stored in the Trie if and only if:

$$root \to s[1] \to ... \to s[n] \to exist = 1$$

And that word's definition vector is stored by:

$$root \to s[1] \to ... \to s[n] \to emptyWord$$

Main Functions:

All functions below perform based on traversing algorithm. They iterate through nodes that correspond to a given string's characters to reach the exact node needed.

- **Search for a word:** Find the definitions stored in trie that are held by a string.
- **Add a word:** Add a word object into trie.
- **Delete a word:** Delete a word in trie.
- **Get words with same prefix "s":** Get words with the same string prefix, we can limit the number K of such words to display for user.

Time Complexity (in that order):

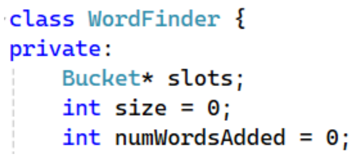
- **O(N)**
- **O(N)**
- **O(N)**
- **O(K*N)**

Run Time:

- Less than 1 second for all those operations.

1.5 Word Finder

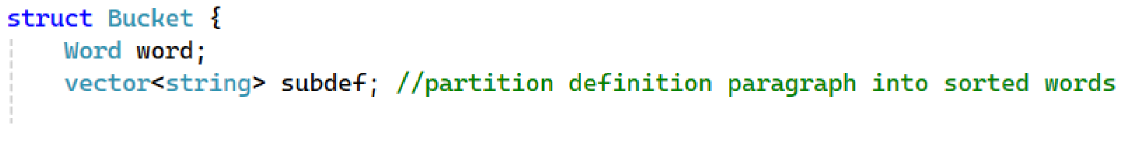
Despite its basic nature, Word Finder proves to be an efficient "Def to Word" searcher.



```
class WordFinder {
private:
    Bucket* slots;
    int size = 0;
    int numWordsAdded = 0;
```

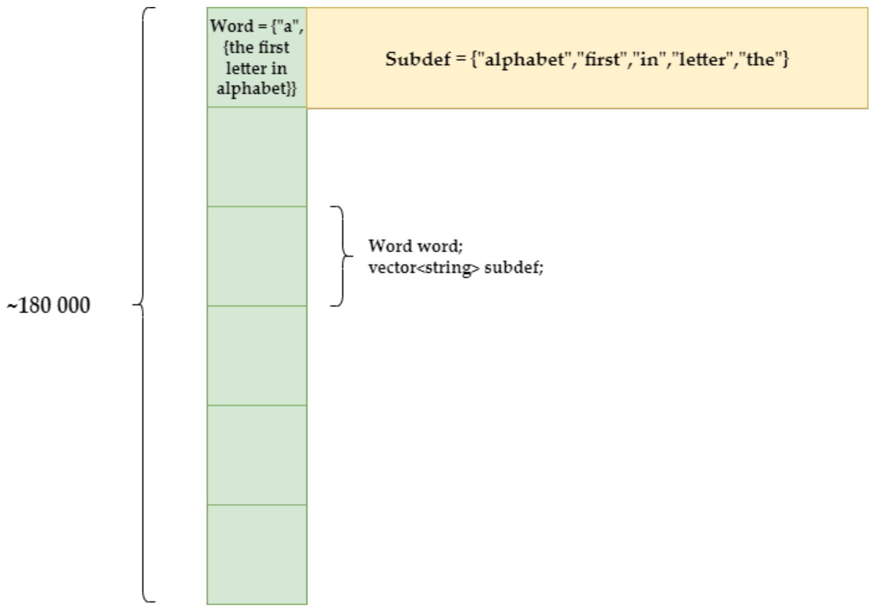
Private:

- **slots**: An array of **Bucket** objects (will be mentioned righ below) with 200000 elements for storing words and potential added words.
- **size**: equals to number of words in a dataset.
- **numWordsAdded**: equals to number of added words in a certain dataset. Sum of the two integers must not exceed 200000.



```
struct Bucket {
    Word word;
    vector<string> subdef; //partition definition paragraph into sorted words
```

- **word**: word object that holds information of a word in our dictionary. This object has only **one** definition (so if a word has 10 definitions, it will be stored across 10 consecutive buckets).


~180 000
Word = {"a", {the first letter in alphabet}}
Subdef = {"alphabet","first","in","letter","the"}
Word word;
vector<string> subdef;

- **subdef**: a **sorted** vector of strings, where each string is a subword of **word**'s definition (we will further explain this below).

Operating Method:

Each of the Bucket from $slots[0] \to slots[size-1]$ represents a word in our dictionary. While from $slots[size] \to slots[size + numWordsAdded - 1]$ corresponds to user-added words.

For easier understanding, take a look at the word *"a"* in the picture above. This word can own many definitions, but each bucket will store one definition only.

The format of **subdef** is a sorted vector string, thus we need to partition our definition *"the first letter in alphabet"* into subwords: { *"the"*, *"first"*, *"letter"*, *"in"*, *"alphabet"*}

Then, we sort those strings in ascending order to obtain: {*"alphabet"*, *"first"*, *"in"*,*"letter"*, *"the"*}

About definition searching:

The most significant application of Word Finder is to find words with a certain definition. More formally, if user type a sentence into searchbar, Word Finder will extract the subwords in that sentence out.

A word is said to be **probable** (which means it will be the answer to user's earlier input) if and only if:

- **All those extract subwords exists in a certain definition of that word.**

For instance, if user types *"first letter in"*, or *"in alphabet the"*, the word *"a"* above is one of the probable answers. Yet if they typed *"a in alphabet"*, it won't match, since the subword *"a"* does not appear in our subdef.

Main Functions:

- **Search Definitions to Word:** Given an input string, find all **probable** words (as mentioned above) with respect to that input.

- **Search Word to Definitions:** Given an input string, search for the word that matches that string.
- **Load a whole dataset:** At the beginning of our program, we will load one dataset in first for further searching.

Time Complexity (in that order):

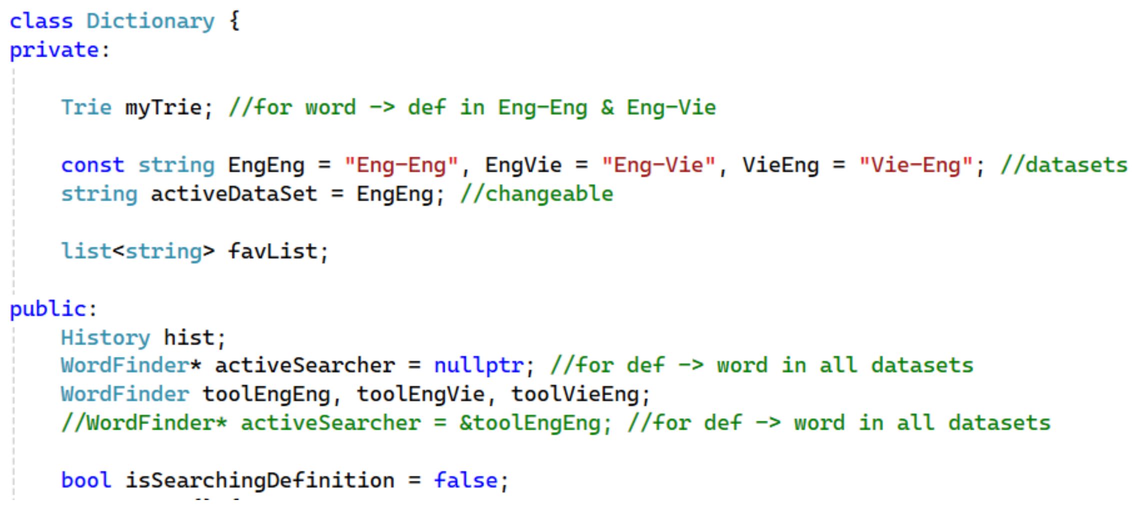
- **O(nmlog(K))** whereas:
  - **n**: number of words in dictionary (about 180000).
  - **m**: number of subwords in input string.
  - **K**: average number of subwords in a definition.
- **O(log(n) + numWordsAdded)** whereas n is number of words in dictionary. The logarithm is for binary searching original words, and the numWordsAdded is iterating the end of Word Finder for user-added words.
- **O(datasetSize)**

Run Time:

- Less than 1 second for the first 2 operations.
- For the last one, 8 seconds for Eng-Eng, 6 and 4 for Eng-Vie and Vie-Eng respectively.

1.6 Dictionary

The aforementioned classes are then packed into a class called Dictionary, which is the most vital tool taking charge of all searching, adding, removing,... actions.



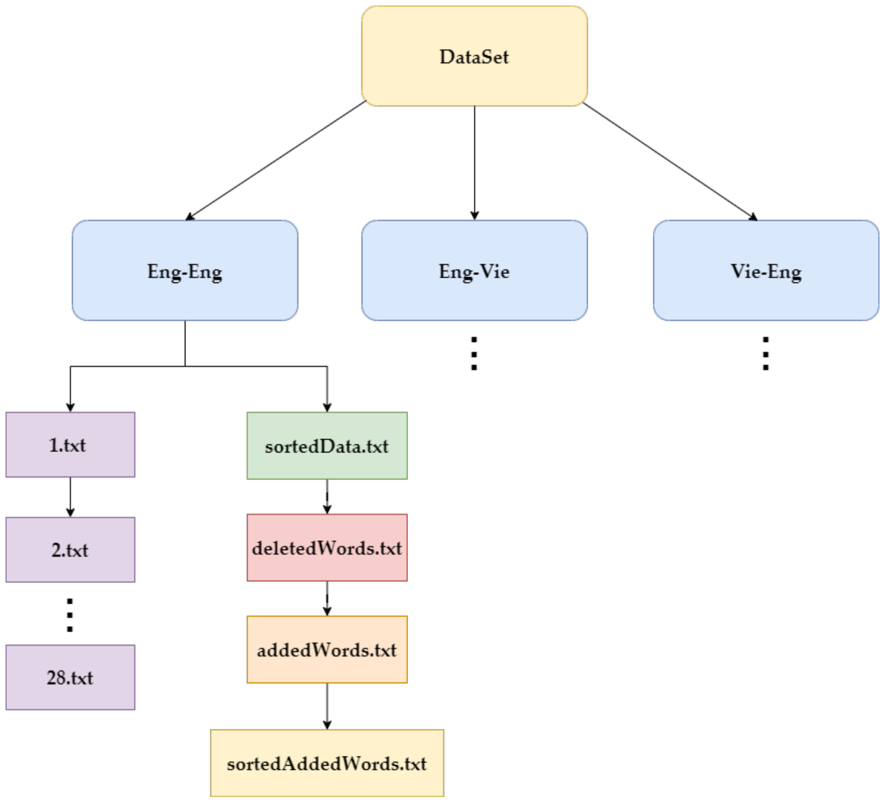
```
class Dictionary {
private:

    Trie myTrie; //for word -> def in Eng-Eng & Eng-Vie

    const string EngEng = "Eng-Eng", EngVie = "Eng-Vie", VieEng = "Vie-Eng"; //datasets
    string activeDataSet = EngEng; //changeable

    list<string> favList;

public:
    History hist;
    WordFinder* activeSearcher = nullptr; //for def -> word in all datasets
    WordFinder toolEngEng, toolEngVie, toolVieEng;
    //WordFinder* activeSearcher = &toolEngEng; //for def -> word in all datasets

    bool isSearchingDefinition = false;
```

2 Data Organisation

In this chapter, we cover the way data is stored in disk and loaded to RAM to search.

2.1 Data Sets

Data Sets are folders than contain words and definitions of our dictionary.



DataSet Folder includes:

- Eng-Eng
- Vie-Eng
- Eng-Vie

Each of the three folders above share similar subfolders and text files, we will just list out one of them.
Eng-Eng folder includes:

- **1 to 28 text files:** Each file represent words that start with a certain initial (apostrophe, hyphen, a,b,c...z). A line within one of the files has the format of:
word (TAB) definition (ENDL).

- **sortedData.txt:** Consists of all words and definitions, this file is read by the Word Finder so it has the format of:
  word (TAB) sorted definition (ENDL).
- **deletedWords.txt:** Stores deleted words in format of:
  word (TAB) definition (ENDL).
- **addedWords.txt:** Stores user-added words in format of:
  word (TAB) definition (ENDL).
- **sortedAddedWords.txt:** Stores user-added words in format identical to **sortedData.txt** to search by definition.

When the program runs, **sortedData.txt** and **sortedAddedWords.txt** will be loaded into Word Finder for definition searching from beginning until closing our app.

- **Time Complexity: O(N)**
- **Run Time: average 6 seconds per folder.**

1 to 28.txt however, will be loaded or unloaded to the Trie accordingly to the user's activities.

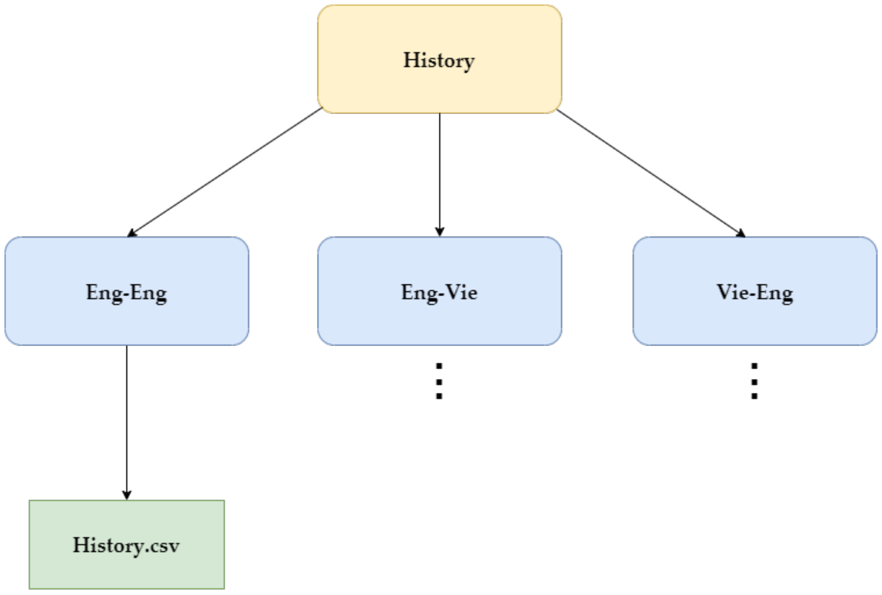
- **Time Complexity: O(N)**
- **Run Time: less than 1 second per txt file.**

At the end of the program, data will be rewrite from RAM back to disk in case changes were made during runtime.

- **Time Complexity: O(N)**
- **Run Time: less than 1 second per folder.**

2.2 History

Format of history data is quite simple, it requires only a csv file to record user's search history.



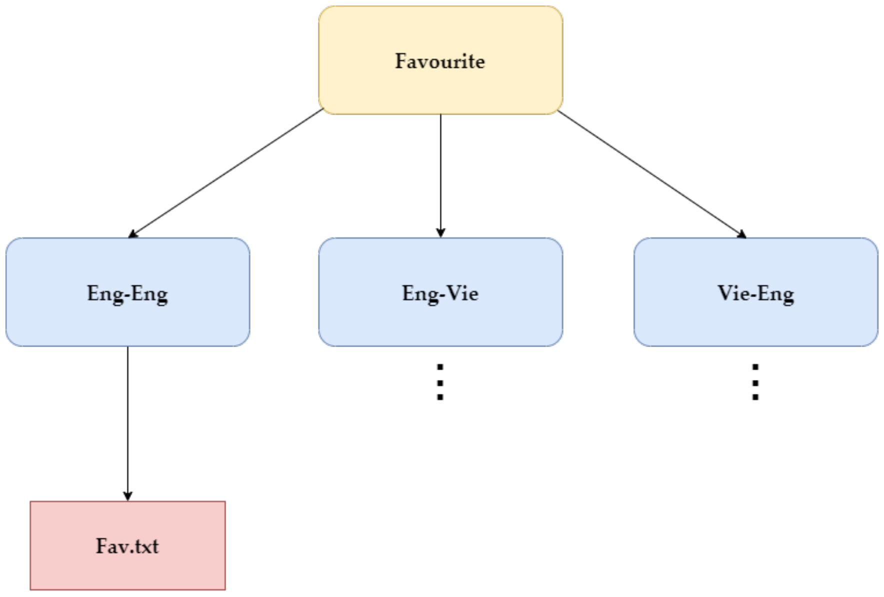
In History.csv, each line includes:

- A column for word.
- A column for time.
- A column for date.

This data then will be loaded onto RAM, and definitions of corresponding words will also be displayed for user.
All functions related to history like: **Showing history, adding a word to history, deleting a word from history,...** cost **O(N)** time complexity and run within 1 second.

2.3 Favourite

User's favourite words are also kept in files and will be displayed with a **highlighted star** when viewing.



In Fav.txt, each line includes:

- A word that was marked as favourite.

All functions related to favourite like: **Showing favourite list, removing a word from favourite, adding to favourite list,...** cost **O(N)** time complexity and run within 1 second.

3 Adding, Removing and Editing Words

In this section, we take a look at adding, removing, editing algorithms and their complexity, runtime.

3.1 Adding New Word

a) Algorithm for adding a word with one definition:

- Check in **deletedWords.txt** to check if there exists such word. If yes then remove that line from the file then ends the algorithm → Word successfully added.
- Otherwise, check if it exists in **Word Finder**. If yes, returns false → Word already existed.
- If no, we then:
  - Add that word to the end of the word finder.
  - Add that word information to **addedWords.txt** and **sortedAddedWords.txt**.

Time Complexity: O(m + log(n)), whereas:

- **m** is size of **deletedWords.txt**.
- **n** is number of words in dataset.

Run Time: less than 1 second.

b) Algorithm for adding a word with multiple definitions:

- Use part a) K times if the word has K definitions

Time Complexity: O(K*(m + log(n))), whereas:

- **K** is the number of definitions.
- **m** is size of **deletedWords.txt**.
- **n** is number of words in dataset.

Run Time: less than 1 second.

3.2 Removing A Word

a) Algorithm for Removing a word with one definition:

- Check if the word exists in **Word Finder**. If no, return false $\rightarrow$ Word does not exists.
- Otherwise, check if that word and its definition appear in **deletedWords.txt**. If yes then return $\rightarrow$ Word was deleted before.
- If not, append the word and its definition to the end of **deletedWords.txt**.

Time Complexity: O(m + log(n)), whereas:

- **m** is size of **deletedWords.txt**.
- **n** is number of words in dataset.

Run Time: less than 1 second.

b) Algorithm for deleting a word with multiple definitions:

- Use part a) K times if the word has K definitions

Time Complexity: O(K*(m + log(n))), whereas:

- **K** is the number of definitions.
- **m** is size of **deletedWords.txt**.
- **n** is number of words in dataset.

3.3 Editing A Definition

The algorithm for editing a word's definition is a little more complicated than the other two:

- First, check if Trie is empty. If yes then skip this step, else edit the exact node that represents that word in the Trie.
- Then, we directly modify the file to change definition. Check if the word appears in **addedWords.txt**, if yes then edit and move on to next step. Otherwise, go to **number.txt** that contains the word and modify.
- Finally, we modify directly on **Word Finder**. When program ends, it will automatically writes **Word Finder** back to **sortedData.txt** and **addedSorted.txt** so every change is up to date.

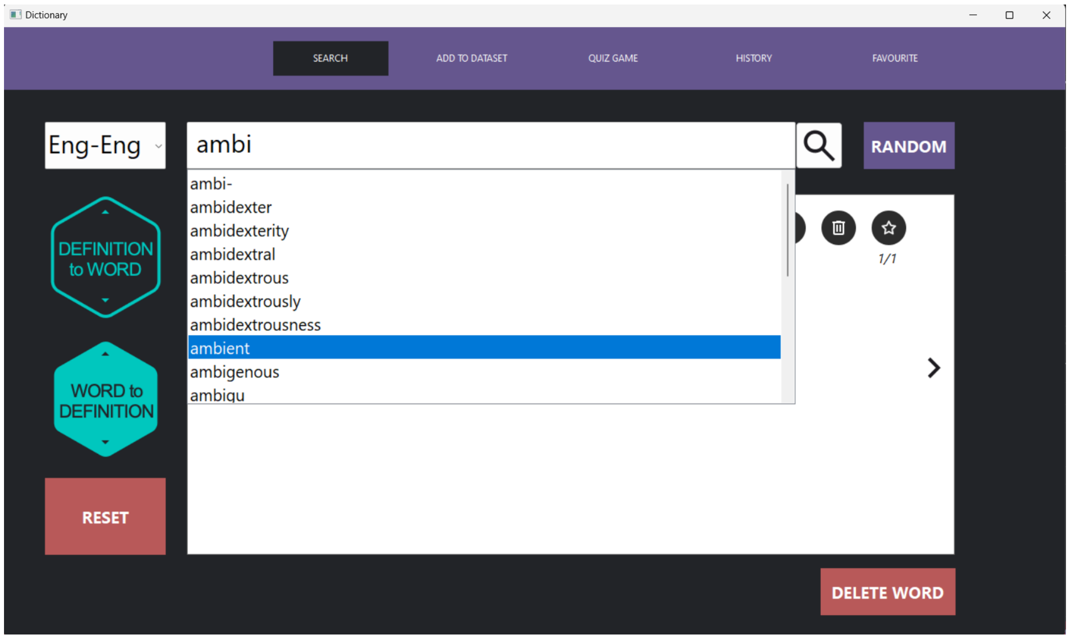
Due to the second step, time complexity is: O(N), whereas:

- **N** is the size of the file **number.txt** that contains the word (maximum 10^6 characters, in the file that stores words starting with *"s"*).

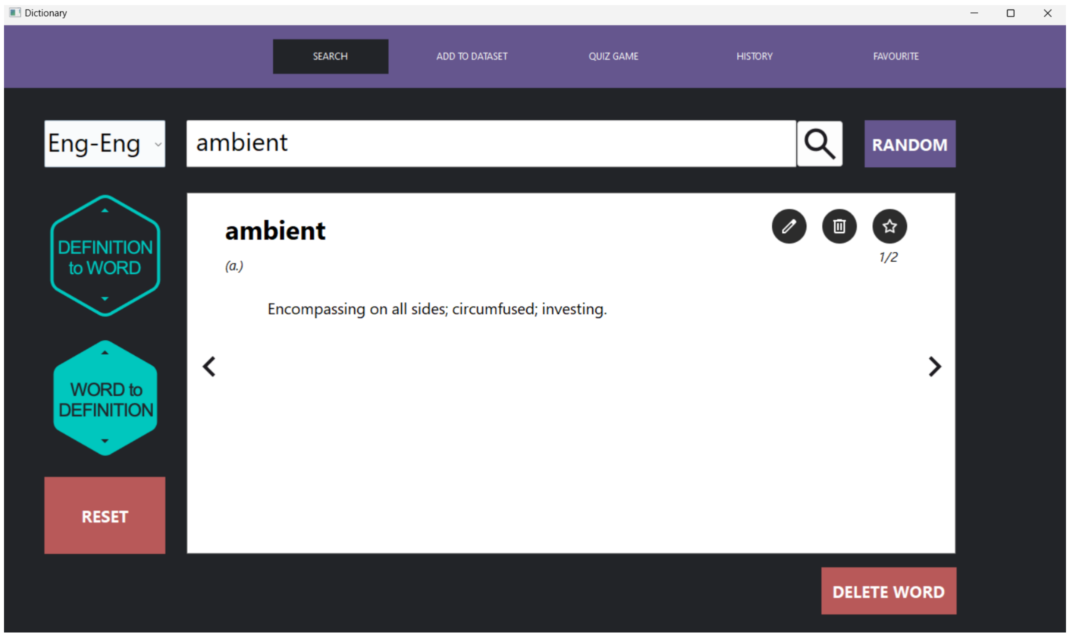
Run time is still less than 1 second, so still good performance!

4 GUI Viewing

4.1 Searching by Word



Viewing that word (can press arrows to move between definitions):

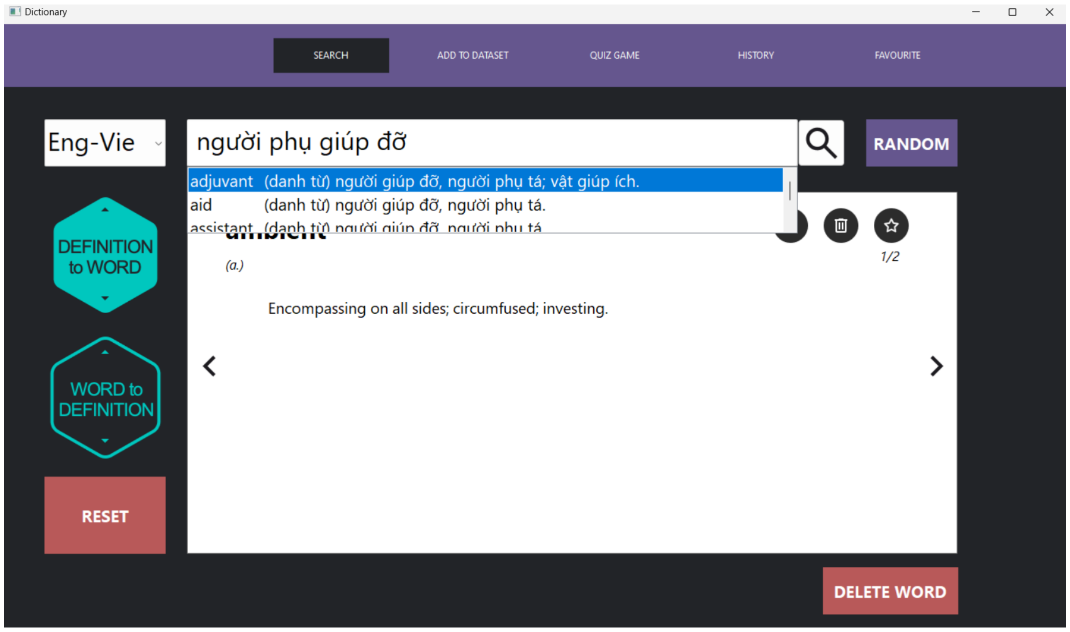


ambient

(a.)

Encompassing on all sides; circumfused; investing.

4.2 Searching by Definition



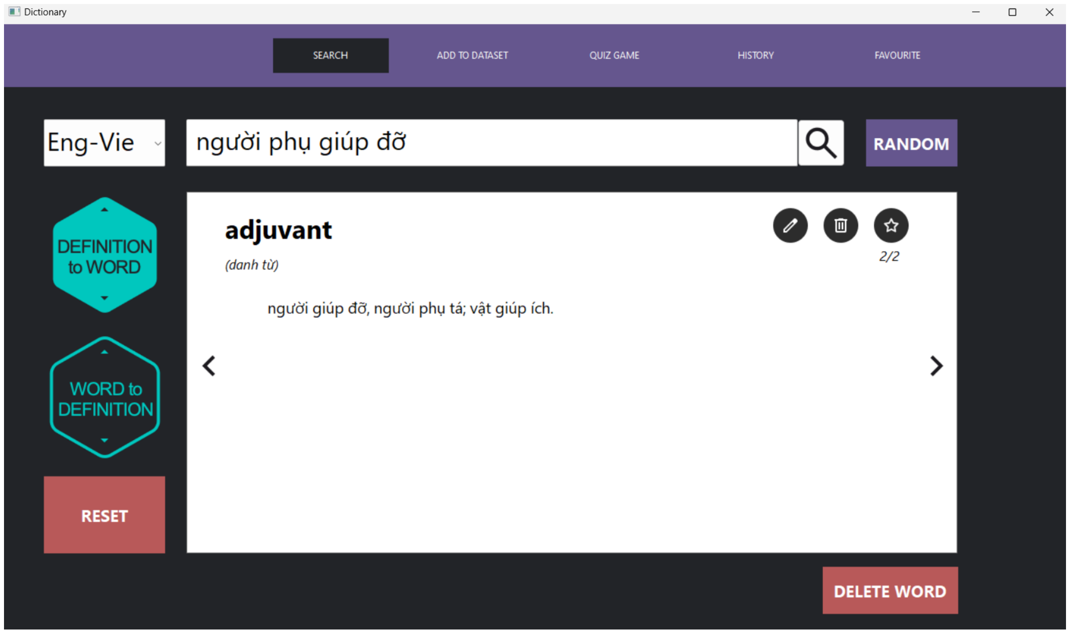
người phụ giúp đỡ

adjuvant (danh từ) người giúp đỡ, người phụ tá; vật giúp ích.
aid (danh từ) người giúp đỡ, người phụ tá.
assistant (danh từ) người giúp đỡ, người phụ tá.

(a.)

Encompassing on all sides; circumfused; investing.

Viewing that word (can press arrows to move between definitions):



người phụ giúp đỡ

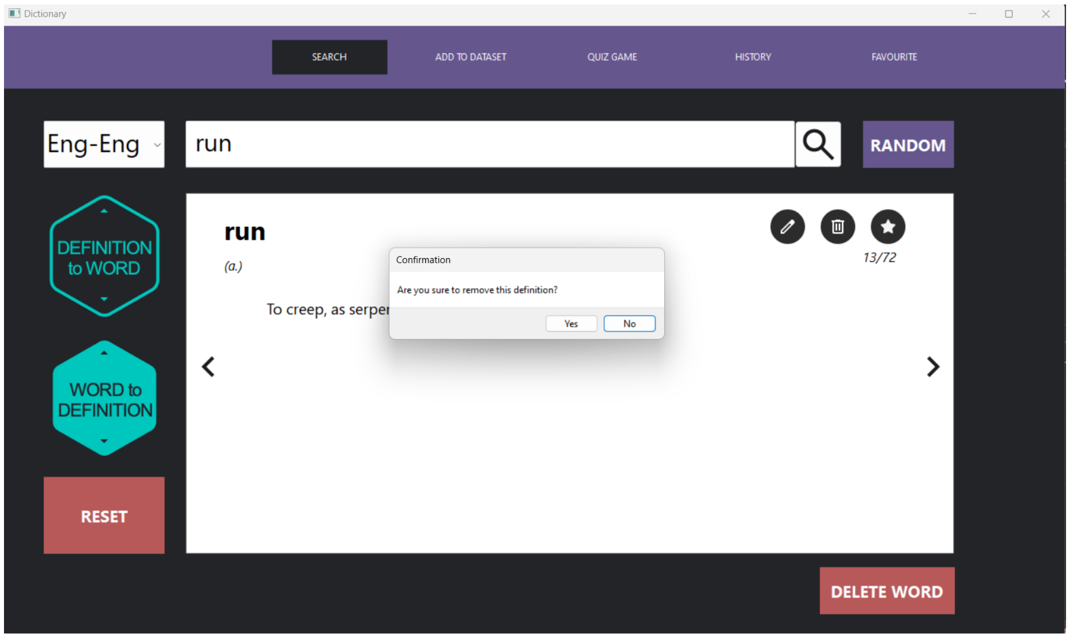
adjuvant

(danh từ)

người giúp đỡ, người phụ tá; vật giúp ích.

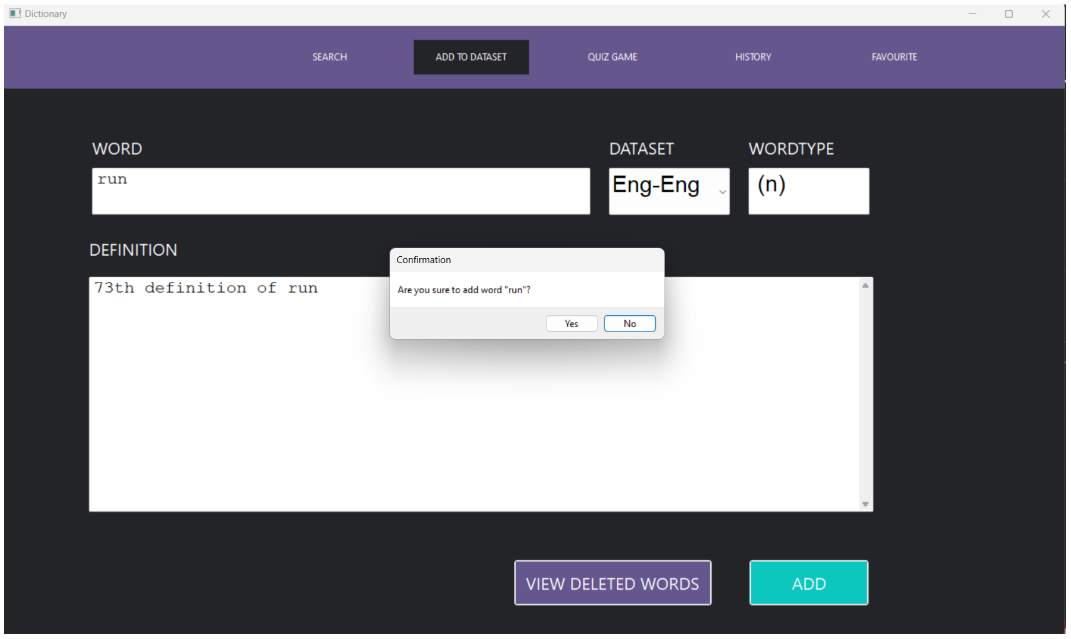
4.3 Deleting Word/Definition

Choose the desired definition by moving back and forth with arrows or change page number. Then, delete one definition using the trash bin (or delete whole word using the red DELETE WORD button)



4.4 Adding Word

Move to "ADD TO DATASET" on navigation bar, we can add a word manually:



run

Eng-Eng

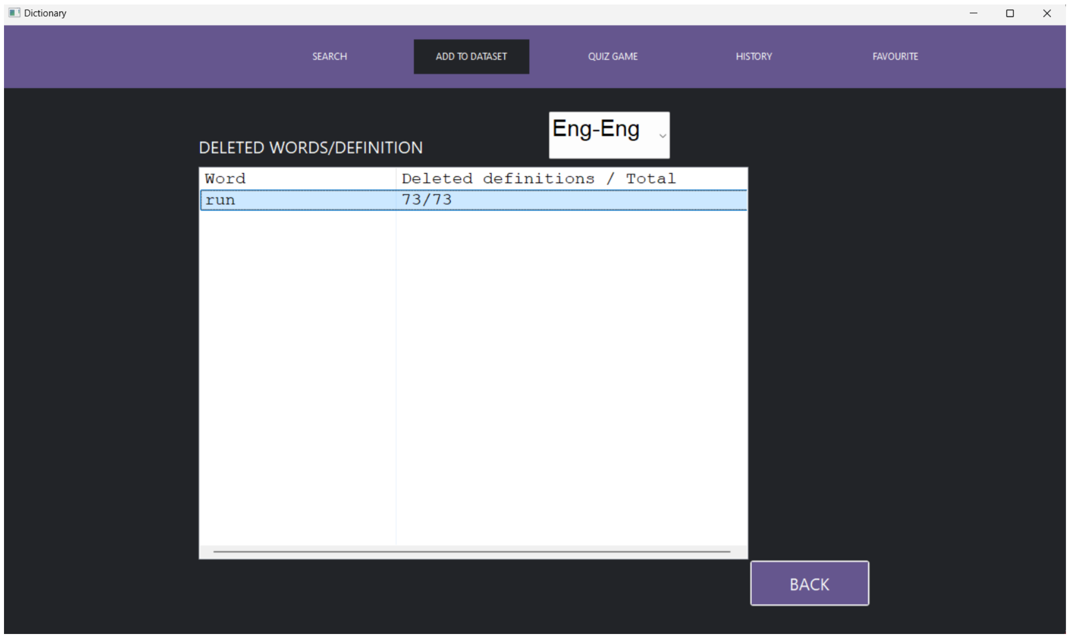
(n)

73th definition of run

Confirmation

Are you sure to add word "run"?

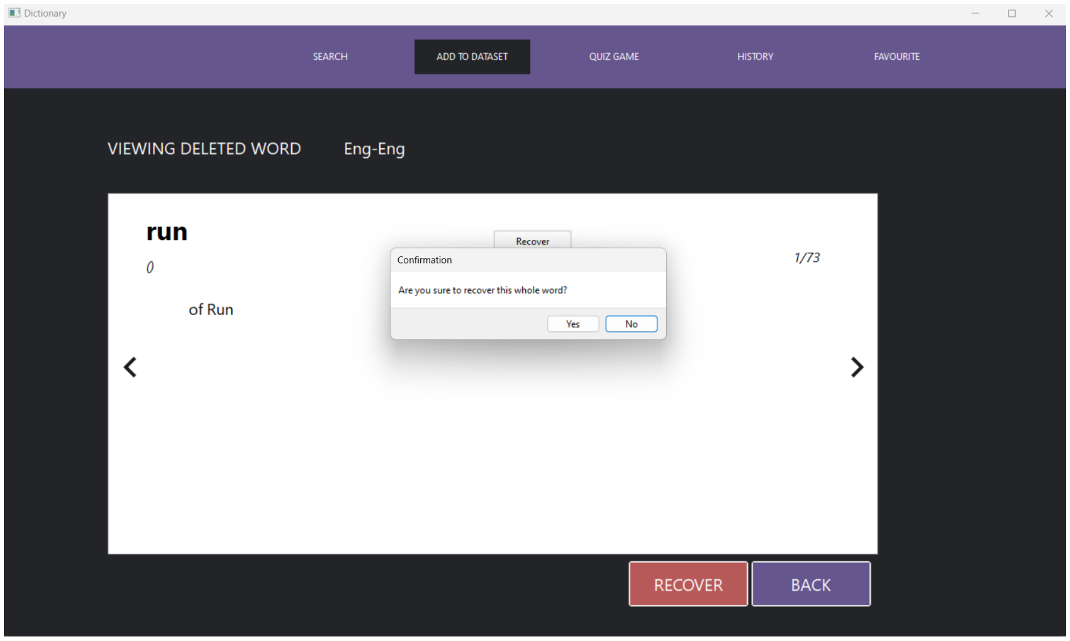
If user wants to add a word-def that was deleted, they can view the deleted words:



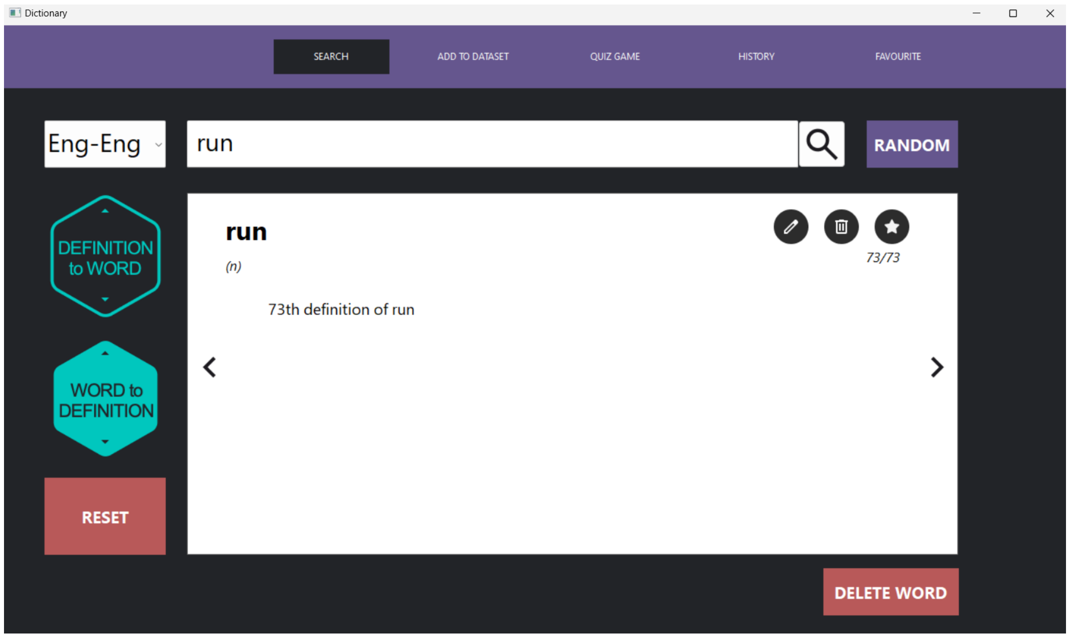
Eng-Eng

Word	Deleted definitions / Total
run	73/73

Then, press recover the definition to re-add it:

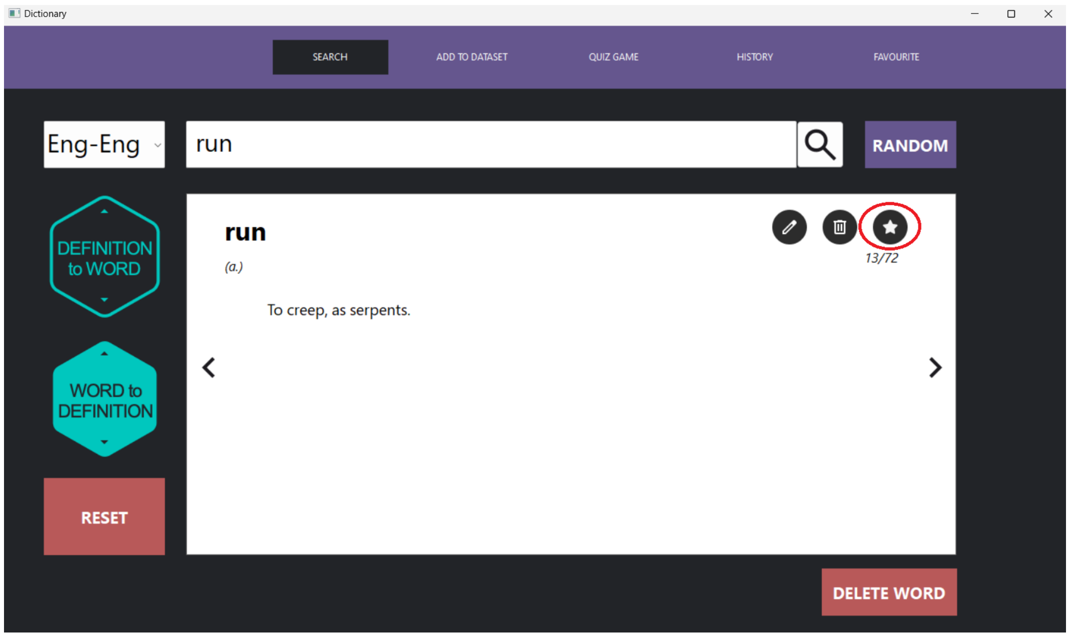


After that, that word now has been added:



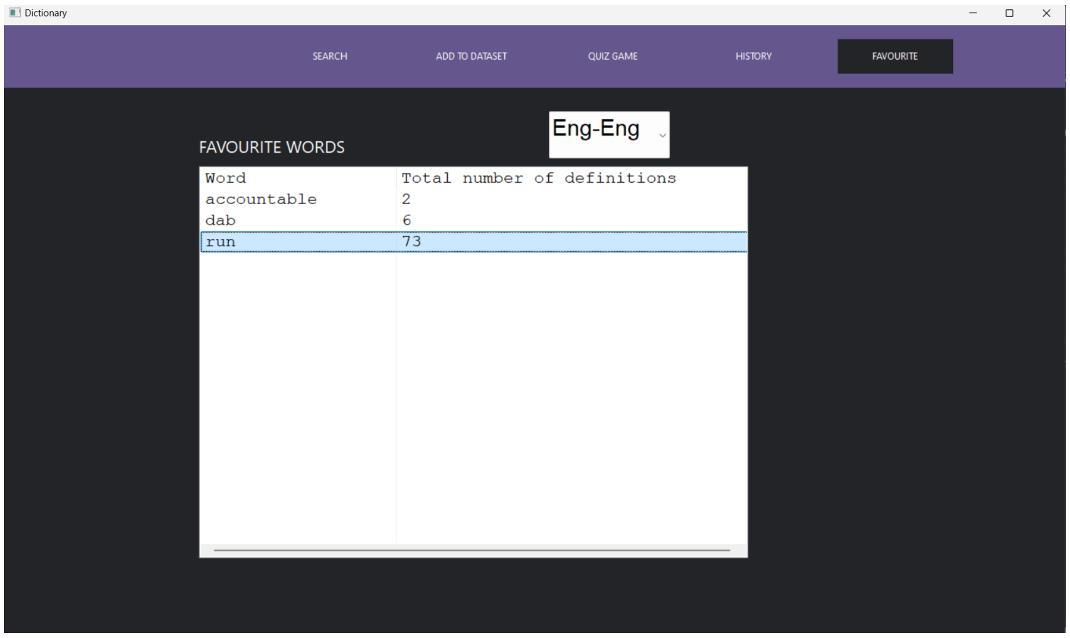
4.5 Favourite

A word can be added to or removed from favourite list by toggle clicking the star button. A white-coloured star implies the word is a favourite one.



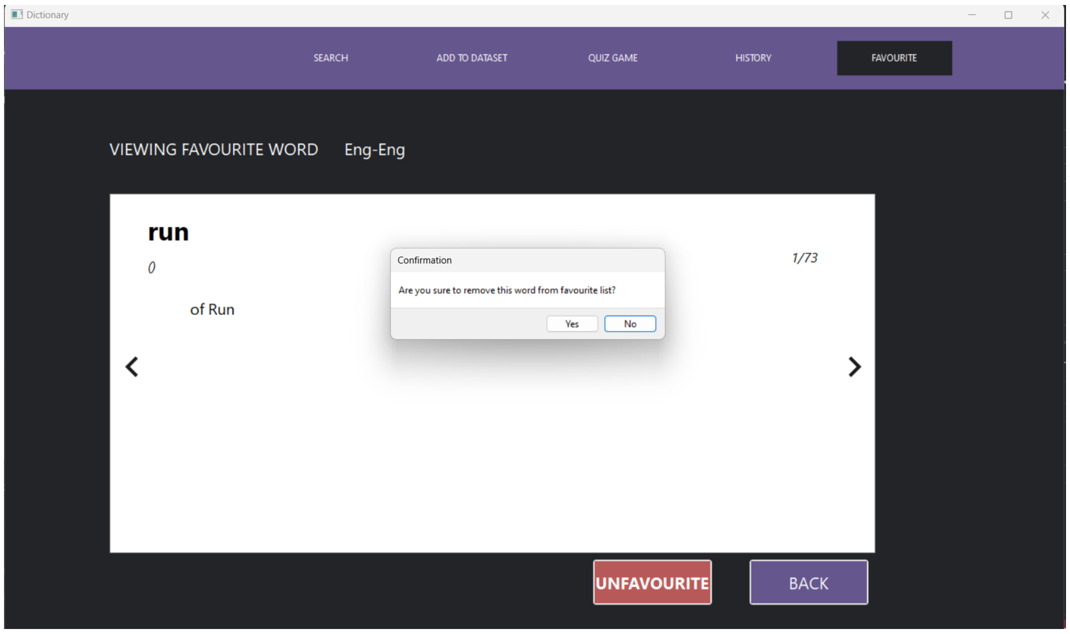
To creep, as serpents.

Move to "FAVOURITE" in the navigation bar, we can view the list of favourite words across different datasets:



Word	Total number of definitions
accountable	2
dab	6
run	73

We can click on one of the words to view. Removing the word from favourite list is also optional:



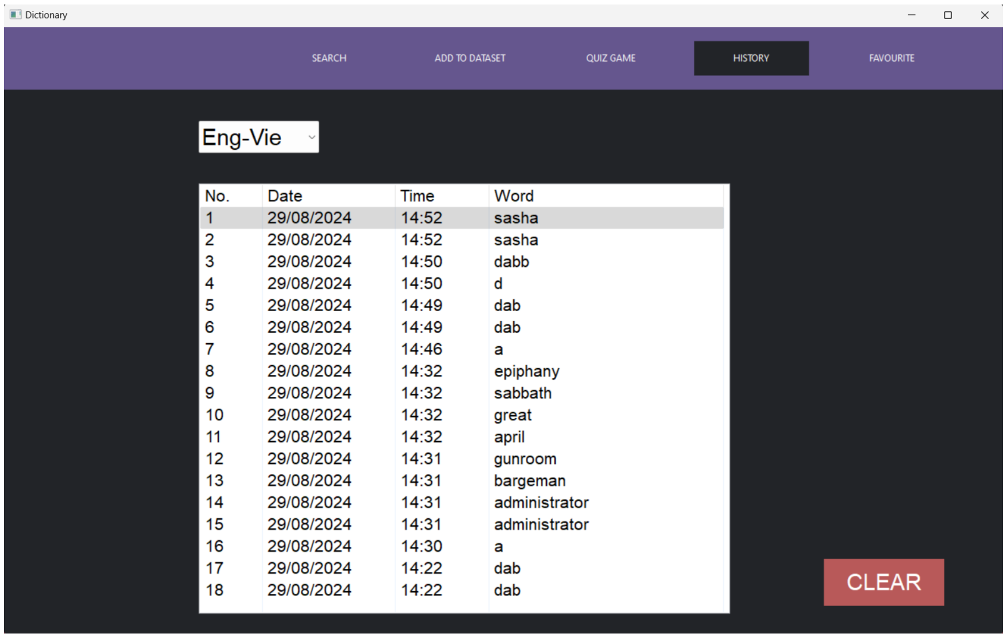
Confirmation

Are you sure to remove this word from favourite list?

Yes No

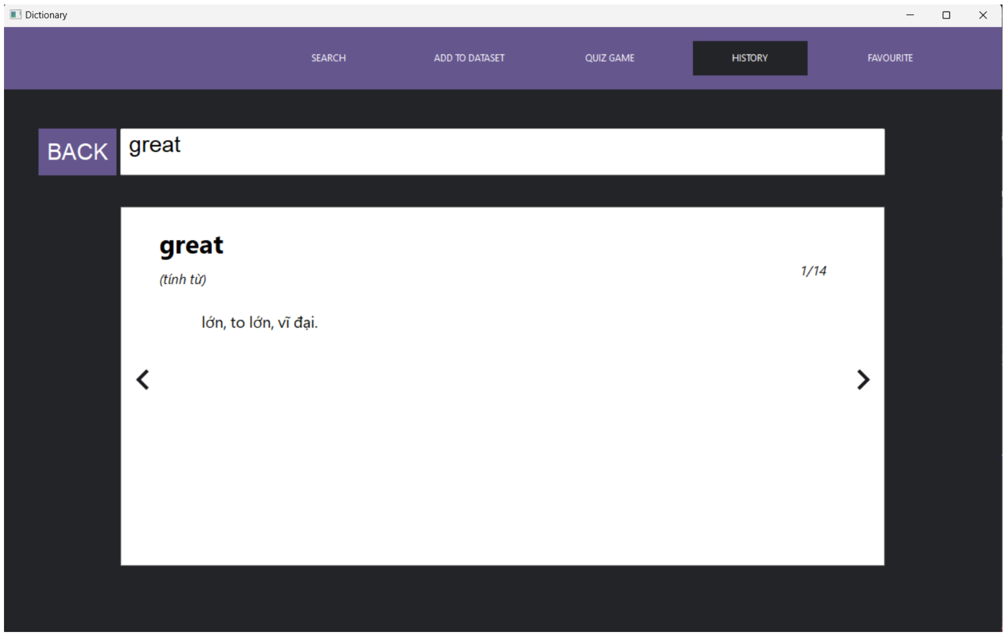
4.6 History

Entering the "HISTORY" on navigation bar, we can choose to view search history of words, including date and time:



No.	Date	Time	Word
1	29/08/2024	14:52	sasha
2	29/08/2024	14:52	sasha
3	29/08/2024	14:50	dabb
4	29/08/2024	14:50	d
5	29/08/2024	14:49	dab
6	29/08/2024	14:49	dab
7	29/08/2024	14:46	a
8	29/08/2024	14:32	epiphany
9	29/08/2024	14:32	sabbath
10	29/08/2024	14:32	great
11	29/08/2024	14:32	april
12	29/08/2024	14:31	gunroom
13	29/08/2024	14:31	bargeman
14	29/08/2024	14:31	administrator
15	29/08/2024	14:31	administrator
16	29/08/2024	14:30	a
17	29/08/2024	14:22	dab
18	29/08/2024	14:22	dab

Left click on one of the words allow you to view all of its definitions:

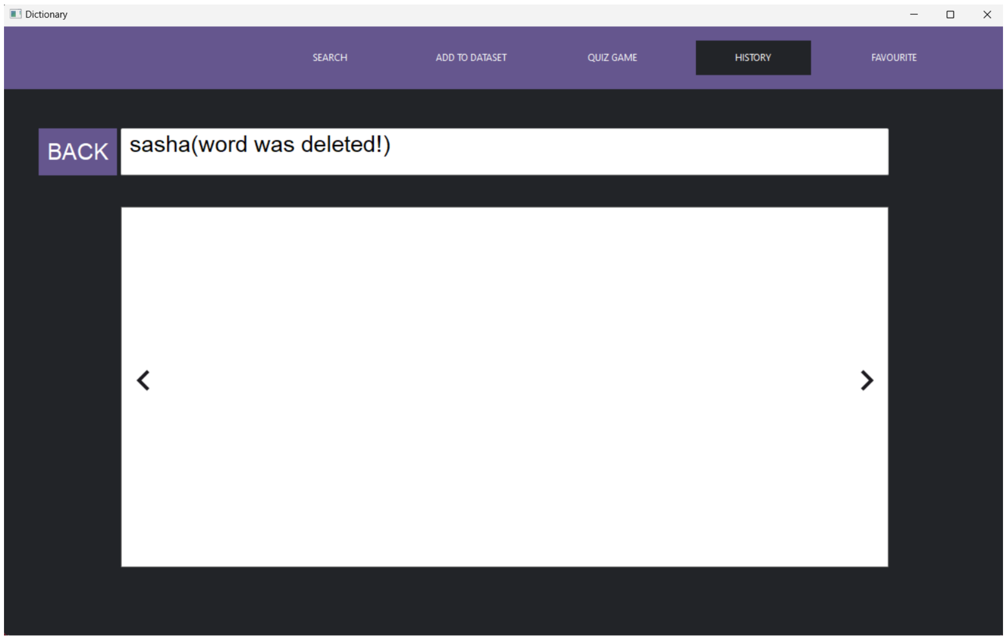


great

(tính từ)

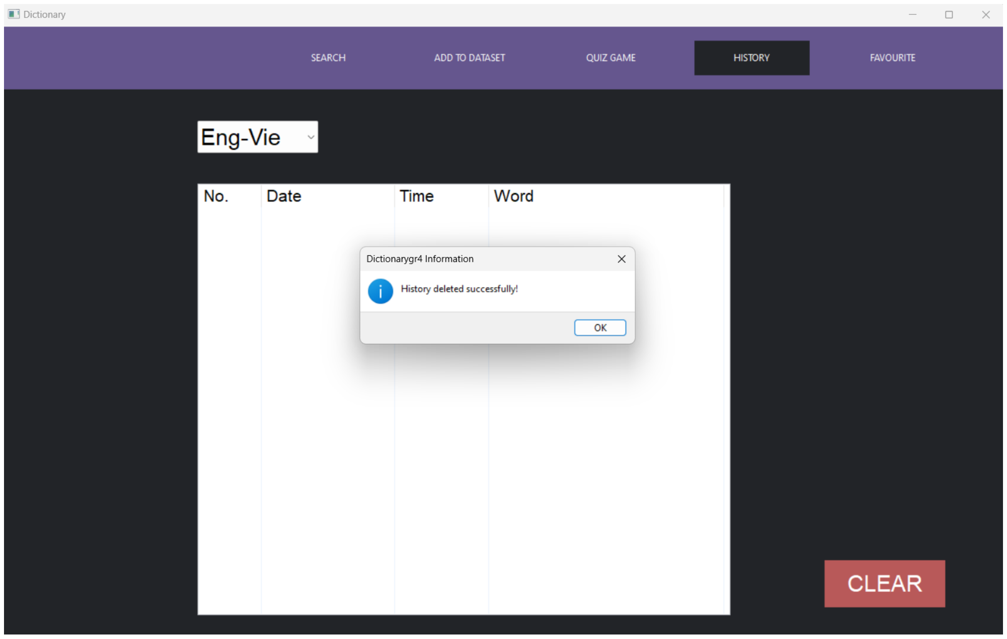
lớn, to lớn, vĩ đại.

However, if that word was deleted by user, history page will note it for you, and display a blank list. To see the full list, you need to re-add it in ”ADD TO DATASET”.



sasha(word was deleted!)

Finally, user can clear the history of one of the datasets:



No.	Date	Time	Word

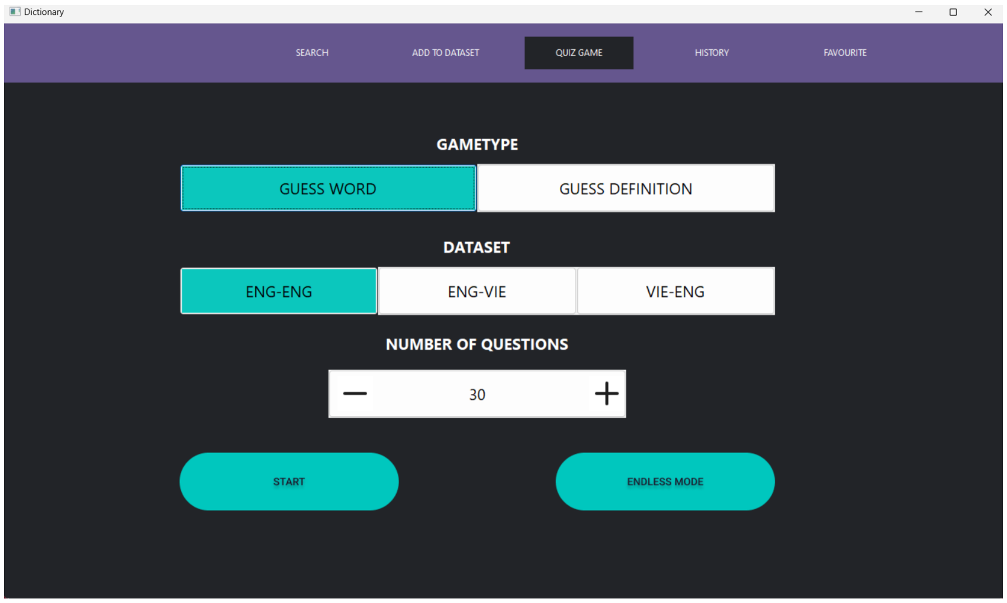
Dictionarygr4 Information

History deleted successfully!

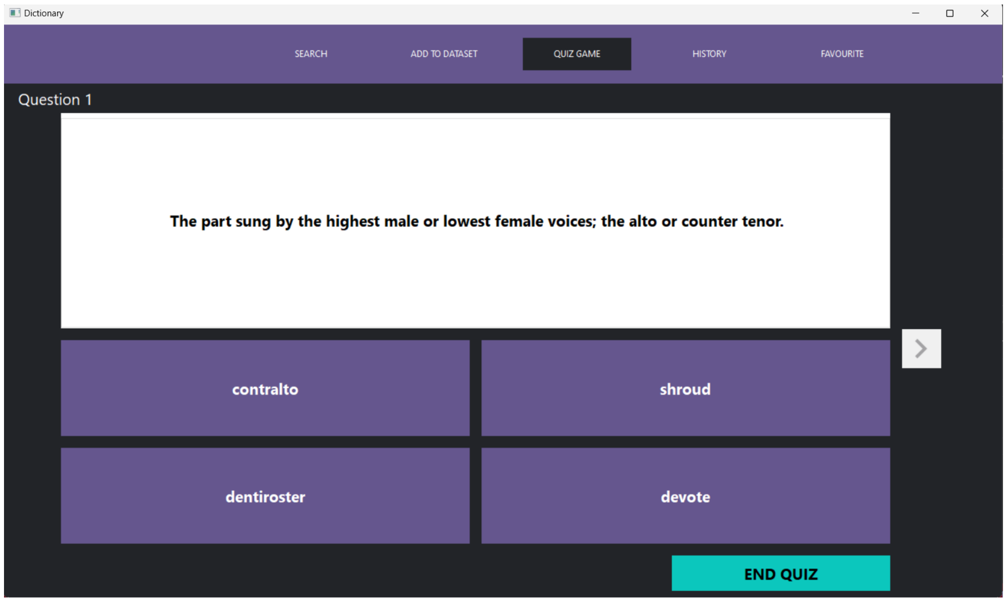
4.7 Quiz

Click on the "QUIZ GAME" button on navigation bar, we will enter the quiz section. User can:

- Choose gamemode.
- Choose dataset.
- Choose number of questions/endless.

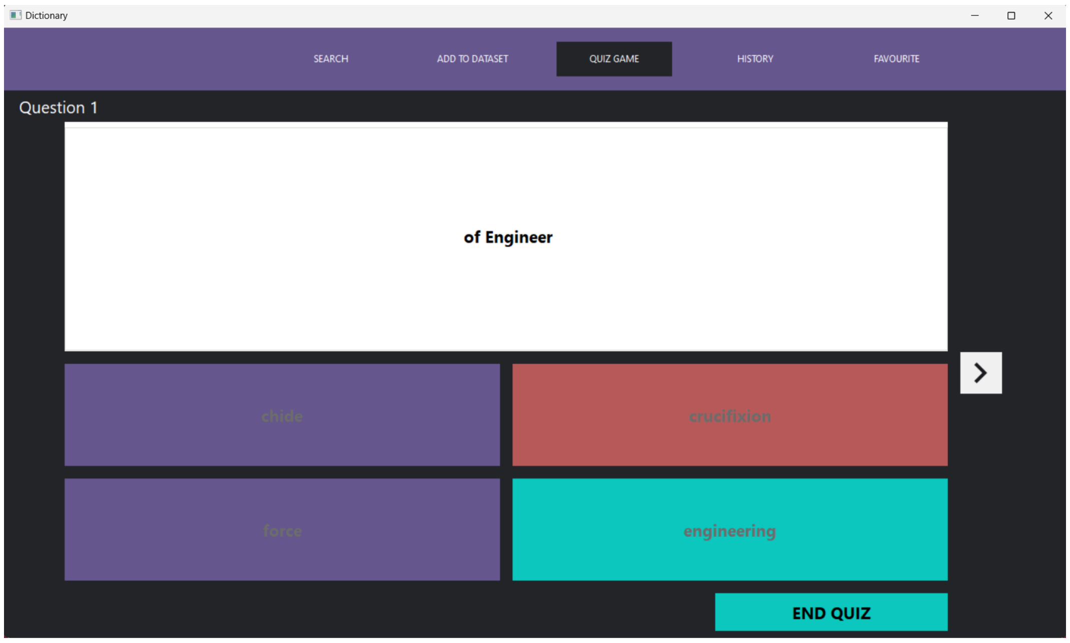


Here is what the game interface looks like:

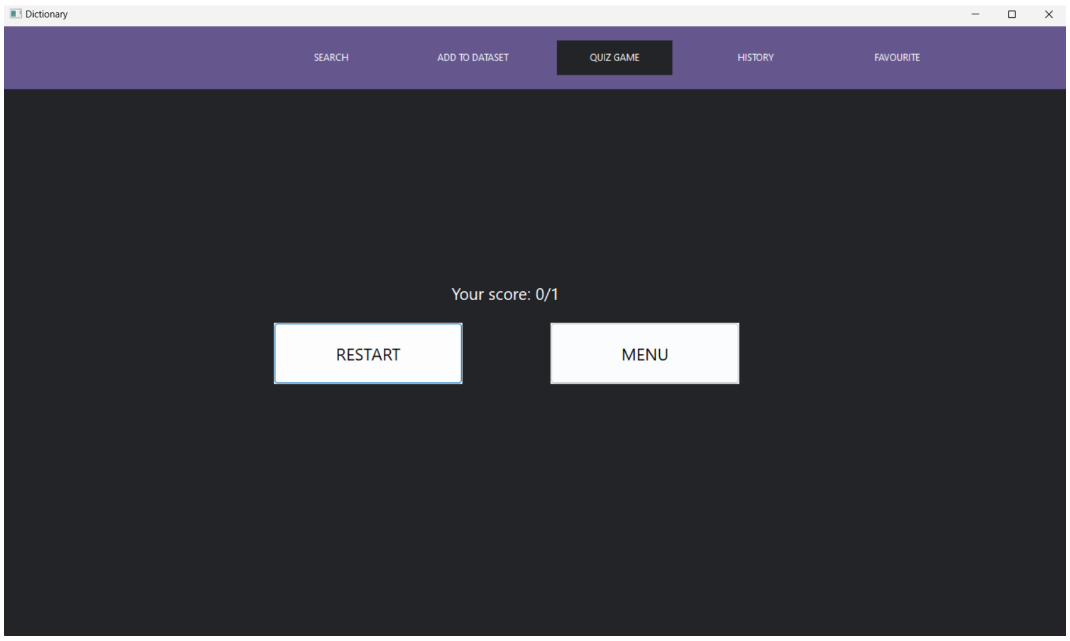


The part sung by the highest male or lowest female voices; the alto or counter tenor.

When user picks the wrong answer:



We can also end the quiz at anytime:


Dictionary
SEARCH
ADD TO DATASET
QUIZ GAME
HISTORY
FAVOURITE
Your score: 0/1
RESTART
MENU