Assignment 2. Structs, 2D arrays, and linked lists

In this assignment your goal is to create a simulation of a *Boggle*-like game. While Boggle is a traditionally multi-player game, this version will be a single-player only. We will add more features in future assignments.

> Boggle is a board word game. The game is played using a plastic grid of lettered dice, in which players attempt to find words in sequences of adjacent letters.
>
> The game begins by shaking a covered tray of 16 cubic dice, each with a different letter printed on each of its sides. The dice settle into a 4×4 tray so that only the top letter of each cube is visible. After the dice settled into the grid, all players simultaneously begin the main phase of play.
>
> The words can be constructed from the letters of sequentially adjacent cells, where "adjacent" are those horizontally, vertically, and diagonally neighboring. Words must be at least three letters long, may include singular and plural (or other derived forms) separately, but may not use the same letter cell more than once per word. There is a time limit at the end of which scores are tallied.
>
> From: https://en.wikipedia.org/wiki/Boggle

See table 1 for dice configuration and table 2 for scoring rules.

Table 1. Letter-dice configuration

Die 0	R	I	F	O	B	X
Die 1	I	F	E	H	E	Y
Die 2	D	E	N	O	W	S
Die 3	U	T	O	K	N	D
Die 4	H	M	S	R	A	O
Die 5	L	U	P	E	T	S
Die 6	A	C	I	T	O	A
Die 7	Y	L	G	K	U	E
Die 8	Q*	B	M	J	O	A
Die 9	E	H	I	S	P	N
Die 10	V	E	T	I	G	N
Die 11	B	A	L	I	Y	T
Die 12	E	Z	A	V	N	D
Die 13	R	A	L	E	S	C
Die 14	U	W	I	L	R	G
Die 15	P	A	C	E	M	D

* Q stands for a diphthong Qu

Table 2. Scoring rules

Word length	Points
3, 4	1
5	2
6	3
7	5
8+	11

Program components

Your program must include the following components:

1. Board generator

Every time a new game starts, the program must generate a new board with accordance to the game rules (see above), i.e. you need to roll and shuffle virtual dice and "place" them in random cells on the grid.

2. Board

The program must display a visualization of the board on the screen throughout the duration of game play. The visualization must resemble grid. The loop for displaying the board can be implemented inside main. The board should be visible to the player during the entire game session. One simple way to ensure this is to clear screen and redraw the board after each new word is entered by the user. One possible approach to solving this problem - using `system("clear")` - is presented in file **`bloop.c`**.

3. User input

The program allows the player to input strings and submit them by pressing the return key, at which point the word checker is invoked.

4. Word checker

The word checker must determine if a given user input conforms to the following rules:

- The submitted word exists in the provided dictionary.
- The word has not been submitted before
- It exists on the board as a path of letters that conforms to the rules of the game. Remember that the same cell cannot be used twice in a single submission.

When the submitted word fails the check, the checker should inform a player of the failure and gives him a possibility to submit another word. If the word passes the check, increment the player's score (refer to Table 2), and inform the player of the value of the submitted word and his new updated score.

5. Session end

Each session lasts as long as needed. When players cannot find any more words, they type 'n' (next) and press a return key. After that the player is presented with his current score, asked for his user name, at which point his name and the current score is submitted to the score board component. And the next session of the game begins.

6. Game end

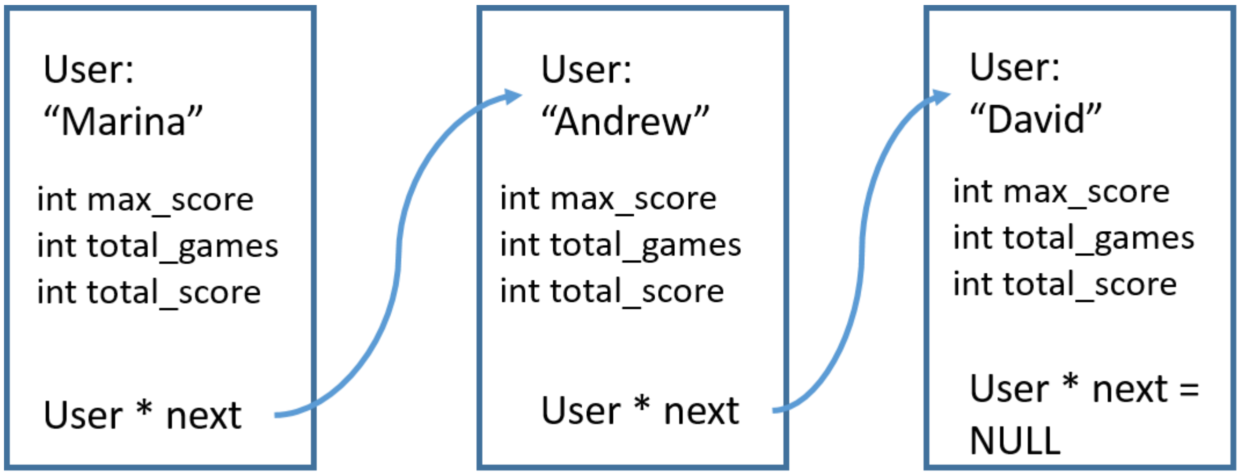
If player types ‘q’ and presses a return key, game terminates. The score board is displayed along with a “game end” message.

7. Score board

The score board is displayed when the game launches and when a game session ends (after the player presses either ‘n’ or ‘q’). In addition to displaying the scores, it should prompt the player to press a button to start a new game session, or to quit.

The score board keeps a list of all players who ever played the game. Each player has an associated best score, number of games played and the total for all games.

Figure 1. Linked list of users



Implementation requirements

1. Each game component should be modularized into a separate set of functions, and should be stored in a separate file. The common information about the current game can be stored in globally-accessible (external) variables.
2. All the code should be compiled using a single *Makefile*, which will compile all your code into a single executable called **boggle**.
3. The user list should be implemented as a ***linked list***.
4. The word checker should use a ***hash table*** to check the words in a provided dictionary and use another hash table to check that the same word is not submitted twice in a single game session. The implementation of the dynamic hash table is provided in 3 files in folder **hash_table_code**. Do not forget that your program should be case-insensitive to the user input and to the words in the dictionary.

Important note: because you are allocating your memory for linked lists and for the hash table dynamically, you have to clean it up after each game session. An example of such a cleanup can be found in function `free_dictionary` in file **dictionary.c**.

5. The program should use stdin, stdout, and stderr for input and output, so that it can be tested with redirecting input and output from/to a file.
6. An optional command-line parameter will specify the input file for the test mode.

So if you run your program:

```
 ./boggle
```

that would mean running in a normal game mode with random board generation and multiple sessions.

If you want to test the correctness, you run:

```
./boggle test1.txt
```

That should read the board definition and the list of words provided in test1.txt, and output the result of a test into file result.txt. You should run the tests for verifying that your program correctly identifies valid and invalid words and assigns the proper total score.

Tests

Test files are provided in archive **tests.tar**.

Each input test file has two lines.

First line is a 16-character string. Every 4 characters represent a single row of the board.

So FEWOSTALJHIVACRM is same as

F	E	W	O
S	T	A	L
J	H	I	V
A	C	R	M

Second line are comma separated words that the test player constructed using the board. *strtok* function might be useful to you to extract each separate word from this line.

Test files explained:

Input:

test1.txt - All words are valid.

test2.txt - Some words are on the board but not from the dictionary. (words STHC,RIVM,ACRM are invalid)

test3.txt - Some words repeat. (words FEW,WAT, MITE, MITE, MITE are invalid)

test4.txt - Some words are in the dictrionary but not on the board. (words SNOW, TABLE, STAIRS are invalid)

test5.txt - Using same letter cell more than once. (words WATT, STALL are invalid)

test6.txt - None of the words are valid. Score should be zero.

Output:

You should output test results into file **result.txt**, inspect its content and compare with the test results below.

The result file will have two lines:

- First line are all words (comma separated and no spaces) that are invalid in the order they appear in the test file. If all words are valid then line should be empty.
- Second line is a total score.

So for example for test1.txt the result.txt should be (first line is empty).

```

12
```

For test2.txt the result.txt should be

```
STHC,RIVM,ACRM
5
```

For test3.txt the result.txt should be

```
FEW,WAT,MITE,MITE,MITE
8
```

Marking scheme

1. Board generator: 15%
2. Board: 10%
3. User input: 15%
4. Word checker: 30%
5. Score board: 20%
6. The code compiles with make, multiple game sessions, memory cleanup after each session: 10%

For a total of 8 points of the course grade.