CSE 220: Systems Fundamentals I

Stony Brook University
Homework Assignment #3
Spring 2024
Due: Sunday, March 31st, 2024 by 9:00 pm EST

Updates to this Document

- Based on the words.txt file, we only have singular words. The word "hybrids" on board06 and “simons” on board08 will be manually added for ease of use. Other than that, adding an "S" should not be permitted unless it creates a new word that is in the words.txt file.

Learning Outcomes

1. An ability to properly manage memory by allocating and deallocating blocks of memory.
2. An ability to work with simple structs.

Assignment Overview

For this assignment you will be implementing functions to support a variant of Upwords. You will not implement the full game, but rather, a set of functions which could be used to implement a full game. Upwords has been around for decades, so there are many tutorials and video demonstrations of how to play it. At the very least, read the PDF linked in this paragraph. The rules given below indirectly refer to the rules.

Now that you have reviewed the official rules, below are the rules we will implement. If you have any questions about what to implement, please ask! We will update this document accordingly.

- The upper-left corner of the game board is located in row #0, column #0.
- A turn consists of placing one or more tiles on the board that connect to an existing word or change an existing word. Any newly created or modified words must be legal. (“Legal” is defined later.) We implement a turn through the `place_tiles` function.
- All letters must be placed in the same row or column.
- All letters placed on the board will be uppercase letters.
- The first word may be placed anywhere on the board.
- The first word must contain at least two letters.
- If a placed word would be legal if the board were bigger and would not create any illegal words, the board will grow horizontally (by adding columns to the right) or vertically (by adding rows on the bottom) to accommodate the new word. The minimum number of rows or columns must be added to accommodate the new word.

- You can stack tiles onto existing tiles to create a new legal word or words. A stack may be at most 5 tiles high.
- When stacking tiles in a turn, you may not cover all of the tiles of an existing word. At least one letter of the original board must remain uncovered and used in the new word.
- A file named `words.txt` will be provided to you with the starter code. Any word appearing in this file is considered "legal".
- We will not implement the rule prohibiting plurals. If adding an S to an existing word creates a new, legal word, then this will be considered a legal move.
    - Based on the words.txt file, we only have singular words. The word "hybrids" on board06 and "simons" on board08 will be manually added for ease of use. Other than that, adding an "S" should not be permitted unless it creates a new word that is in the words.txt file.
- We will not implement drawing tiles from a draw pile or a "rack" of player tiles.
- We will not implement points or scoring.
- We will not implement the rule prohibiting capitalized words. Any word appears in `words.txt` is considered legal. All letters appear in uppercase letters on the board.
- We will not implement the concept of challenging a word.
- We will not implement the concept of exchanging a letter because we will not implement the concept of a rack of player tiles.
- We will not implement any kind of game-ending conditions.

Required Features and Functionality

To support the playing of our version of Upwords, you must implement the data structure functions described below. Function stubs are provided in `hw3.c`. Feel free to create whatever helper functions you like. Define the struct in the provided `hw3.h` file. Only `src/hw3.c` and `include/hw3.h` will be pulled out of your GitHub repository for grading.

The `GameState` struct

As part of this assignment you are required to design and implement a data structure with the named `GameState`. The functions you are required to implement rely on the existence and usage of this data structure. An empty struct has been created for you in `hw3.h` to fill in.

Functions to Implement

`GameState* initialize_game_state(const char *filename)`

This function opens the indicated files, whose contents represents the state of a valid game, and initializes a new `GameState` struct with the state of the game. All stacks of tiles are assumed to be 1 tile high. An example board file is shown below. Assume that the last line of the file contains a newline. You may assume that the input file exists and specifies a valid game state. A period indicates a space with no tile on it.

```
..........
..........
...T.P....
..HOME....
...P.T....
...M.A....
...A.L....
...N......
```

GameState* place_tiles(GameState *game, int row, int col, char direction, const char *tiles, int *num_tiles_placed)

This function attempts to place the tiles given in the `tiles` argument. Assume that `tiles` is null-terminated and consists only of uppercase letters and spaces. The `direction` argument indicates whether the placement is vertically downward (`direction = 'V'`) or horizontally to the right (`direction = 'H'`). The `row` and `col` arguments indicate where the first tile of the tiles must be placed. `row` and `col` are 0-based indices. Before returning, write the number of tiles placed in `*num_tiles_placed` (e.g., `*num_tiles_placed = some_variable;`)

A space character in the `tiles` argument indicates that the user will use an existing tile on the board to create one or more valid words. For example, consider the following state of the board:

```
..........
..........
...T.P....
..HOME....
...P.T....
...M.A....
...A.L....
...N......
```

Suppose `row = 6, col = 1`, `tiles = "SN I"`, `direction = 'H'`. This would be a valid move because we will create the word SNAIL by placing these three tiles. The value of `*num_tiles_placed` would need to be set to 3 before the function returns. The updated board is shown below:

```
..........
```

```
..........
...T.P....
..HOME....
...P.T....
...M.A....
.SNAIL....
...N......
```

The function returns a pointer to a `GameState` struct that represents the updated state of the game. If the tiles cannot be placed for some reason, the function must make no changes to the current state of the game and simply returns `game`. The following are reasons why the given tiles cannot be placed:

- Any of the rules given under “Assignment Overview” are violated.
- `row` or `col` are invalid (i.e., `row` and/or `col` are less than 0; `row` ≥ the number of rows; `col` ≥ the number of columns)
- `direction` is not one of `'H'` or `'V'`

Example #1: Adding tiles requires increasing the width of the board

Game state before the function call `place_tiles(game, 4, 6, 'H', "ABLEWARE", &num)`:

```
..........
..........
...T.P....
..HOME....
...P.T....
...M.A....
...A.L....
...N......
```

Game state after the function call:

```
..............
..............
...T.P........
..HOME........
...P.TABLEWARE
...M.A........
...A.L........
...N..........
```

Example #2: Adding tiles requires increasing the width of the board

Game state before the function call `place_tiles(game, 2, 8, 'H', " ROID", &num)`:

```
.........
.........
......AND
.........
```

Game state after the function call:

```
.............
.............
......ANDROID
.............
```

`The function call place_tiles(game, 2, 9, 'H', "ROID", &num)` should be invalid, since 9 exceeds the maximum column number in the original board.

Example #3: Adding tiles requires increasing the height of the board

Game state before the function `place_tiles(game, 3, 12, 'V', "P OMENADE", &num)`:

```
..............
..............
...T.P........
..HOME........
...P.TABLEWARE
...M.A........
...A.L........
...N..........
```

Game state after the function call:

```
..............
..............
...T.P........
..HOME......P.
...P.TABLEWARE
...M.A......O.
...A.L......M.
...N........E.
............N.
```

```
............A.
............D.
............E.
```

GameState* undo_place_tiles(GameState *game)

The undo_place_tiles function undoes the effects of the most recent successful call to places_tiles_function. By “successful” we mean a call to place_tiles that actually placed tiles on the board. If there is no previous successful call to place_tiles, a call to the undo_place_tiles has no effect. The function returns a pointer to GameState structure representing the updated (i.e. previous) state of the game. If the undo fails because there is no previous game state to revert to, the function simply returns game.

An example will help to demonstrate how this function should work.

```
int num_tiles_placed;
GameState *game = initialize_game_state("./tests/boards/board01.txt");
```

Initial state of the board:

```
..........
..........
..........
..HOME....
..........
..........
..........
..........
```

Let’s place a few sets of tiles successfully:

```
game = place_tiles(game, 2, 3, 'V', "T PMAN", &num_tiles_placed);
game = place_tiles(game, 2, 5, 'V', "P TAL", &num_tiles_placed);
game = place_tiles(game, 6, 1, 'H', "SN I", &num_tiles_placed);
```

The state of the game as represented by the game struct is:

```
..........
..........
...T.P....
..HOME....
...P.T....
```

```
...M.A....
.SNAIL....
...N......
```

Now suppose we call `undo_place_tiles`: `game = undo_place_tiles(game);`

The game state (as represented by `game`) is now:

```
..........
..........
...T.P....
..HOME....
...P.T....
...M.A....
...A.L....
...N......
```

We call it again: `game = undo_place_tiles(game);`

The game state (as represented by `game`) is now:

```
..........
..........
...T......
..HOME....
...P......
...M......
...A......
...N......
```

Another example. The height of each pile is given below the state of the board. Note how the heights increase when tiles are placed, and decrease when the moves are later undone:

```
GameState *game = initialize_game_state("./tests/boards/board2x2.txt");
game = place_tiles(game, 0, 0, 'V', "ABDOMINOPOSTERIOR",
&num_tiles_placed);
game = place_tiles(game, 2, 0, 'H', " INER", &num_tiles_placed);
game = place_tiles(game, 2, 4, 'H', "URIC", &num_tiles_placed);
game = place_tiles(game, 1, 5, 'H', "OH", &num_tiles_placed);
game = place_tiles(game, 3, 7, 'V', "ARE", &num_tiles_placed);
game = place_tiles(game, 5, 4, 'H', "TAK", &num_tiles_placed);
game = place_tiles(game, 5, 4, 'H', "B   D", &num_tiles_placed);
```

State of the board (17 rows, 9 columns):

```
A........
B....OH..
DINEURIC.
O......A.
M......R.
I...BAKED
N........
O........
P........
O........
S........
T........
E........
R........
I........
O........
R........
100000000
100001100
111121110
100000010
100000010
100021111
100000000
100000000
100000000
100000000
100000000
100000000
100000000
100000000
100000000
100000000
100000000
```

Call `undo_place_tiles` twice. The state of the board (17 rows, 8 columns) is now:

```
A.......
B....OH.
DINEURIC
O......A
M......R
```

```
I......E
N.......
O.......
P.......
O.......
S.......
T.......
E.......
R.......
I.......
O.......
R.......
10000000
10000110
11112111
10000001
10000001
10000001
10000000
10000000
10000000
10000000
10000000
10000000
10000000
10000000
10000000
10000000
10000000
```

If your program was initialized (via `initialize_game_state`) with a board already containing tiles, the `undo_place_tiles` function cannot and must not remove any of those tiles.

`void free_game_state(GameState *game)`

This function deallocates all memory associated with a game of Upwords. This includes any and all data structures needed by any required or helper functions, including any data structures needed to implement the undo operation.

`void save_game_state(GameState *game, const char *filename)`

This game saves to disk in the named file the current state of a game of Upwords represented by the game argument. The saved file must show not only the top tile of each stack, but the height of each stack. Include a newline at the end of the last line of the file. The example below shows a sample file (`expected_outputs/multiple06.txt`):

```
A..........
B....OH....
DINEURIC...
O......A...
M......B...
I....PABBLE
N......A...
O......G...
P......E...
O..........
S..........
T..........
E..........
R..........
I..........
O..........
R..........
10000000000
10000110000
11112111000
10000001000
10000001000
10000531321
10000001000
10000001000
10000001000
10000000000
10000000000
10000000000
10000000000
10000000000
10000000000
10000000000
10000000000
```

Dynamic Memory Allocation and Deallocation is Required

Your program must manage the memory of your data structures (including arrays) dynamically. Your code may not contain any statically allocated data structures (i.e., arrays with fixed sizes) except for simple, routine tasks like searching the `words.txt` file. In such cases you may create a `char` array of fixed size to help you search the text file. If you are not sure whether your code is acceptable according to this assignment requirement, please ask.

A solution that otherwise makes use of statically-allocated data structures will be penalized at least 50% of the maximum credit.

No Hard-coding of Boards is Allowed

We noticed a significant amount of hard-coding in Homework Assignment #2. Any hard-coding of board names, board sizes, etc. will result in a zero for the assignment. To combat hard-coding, board filenames on CodeGrade will be automatically renamed on disk and in test cases before execution.

Template Repository

Starter code is available on CodeGrade and GitHub, as with previous assignments.

Building and Running Your Code at the Command Line

1. Configure the code to build. This needs to be done only once:
   ```
   cmake -S . -B build
   ```
2. Build the code after changing the code:
   ```
   cmake --build build
   ```
3. Run the test cases.
   ```
   ./build/run_all_tests
   ```

Testing & Grading Notes

More test cases (both visible and hidden) might be added after the assignment's release in response to students' questions.

Note: any evidence of hard-coding test case input or output will automatically result in a score of zero for the assignment. This kind of behavior is borderline academic dishonesty.

Generative AI

Generative AI may not be used to complete this assignment.

Academic Honesty Policy

Academic honesty is taken very seriously in this course. By submitting your work for grading you indicate your understanding of, and agreement with, the following Academic Honesty Statement:

1. I understand that representing another person's work as my own is academically dishonest.
2. I understand that copying, even with modifications, a solution from another source (such as the web or another person) as a part of my answer constitutes plagiarism.
3. I understand that sharing parts of my homework solutions (text write-up, schematics, code, electronic or hard-copy) is academic dishonesty and helps others plagiarize my work.
4. I understand that protecting my work from possible plagiarism is my responsibility. I understand the importance of saving my work such that it is visible only to me.
5. I understand that passing information that is relevant to a homework/exam to others in the course (either lecture or even in the future!) for their private use constitutes academic dishonesty. I will only discuss material that I am willing to openly post on the discussion board.
6. I understand that academic dishonesty is treated very seriously in this course. I understand that the instructor will report any incident of academic dishonesty to the University's Academic Judiciary.
7. I understand that the penalty for academic dishonesty might not be immediately administered. For instance, cheating on a homework assignment may be discovered and penalized after the grades for that homework have been recorded.
8. I understand that buying or paying another entity for any code, partial or in its entirety, and submitting it as my own work is considered academic dishonesty.
9. I understand that there are no extenuating circumstances for academic dishonesty.