CMPS 12B, Fall 2018: HW 3
Making some moves

- All assignments must be submitted through git. Please look at the Piazza guide on submitting assignments.
- Follow instructions, and carefully read through the input/output formats.
- Clearly acknowledge sources, and mention if you discussed the problems with other students or groups. In all cases, the course policy on collaboration applies, and you should refrain from getting direct answers from anybody or any source. If in doubt, please ask the instructors or TAs.

1 Problem description

Main objective: Store a chessboard as a linked list of chess pieces, implement moves, and determine if king is under attack. You cannot use built in libraries for manipulating data. No arraylists, no hash tables, etc.Assume an 8×8 chessboard. Implement a procedure that, given a chessboard, makes a series of moves. Let squares be indexed as (column,row) pairs. Given a source square (x, y) and a destination square (x', y'), determine if the piece (if any) at (x, y) can legally move to (x', y'). Given a sequence of moves (x_1, y_1) to (x'_1, y_1), (x_2, y_2) to (x'_2, y'_2), etc., implement all these moves to determine the final chessboard.

1.1 What is a legal move?

- Color alternates: in the sequence of moves, assume that white plays first, then black, then white, etc.
- Non-empty source: obviously, there must a piece to move at the source square.
- Piece moves according to its rules: each piece moves according to certain rules. A move is valid only if it moves the respective piece appropriately. *In this assignment, you do not need to worry about unconventional moves like castling, en passant, or pawns promoted by reaching the last row.*
- Destination occupied by piece of same color: if the destination square has a piece, it must be of a different color (this is a capture).
- Path is blocked: when a move is performed, there should be no other piece (of any color) in its "path". This is not true for knights, which can "jump" over any pieces in its path.

- King cannot be in check after move: a king is in check, if it can be attacked by a piece of the opposite color. According to the rules of chess, a king can never end up in check. Suppose white moves. At the end of the move, the white king is in check. Then, this move is invalid. (Indeed, if white has no move that prevents check, then white has lost.) This is the hardest condition to handle, so save this for last while coding.

Thus, given a sequence of moves, you have to determine if the *sequence of moves* is legal. Note that each move changes the board, so you have determine legality with respect to the current board.

1.2 Suggestions for coding:

Clearly, the solution of the previous assignment can be used. You can simply copy the code of your previous HW2, or use the solution for HW2 that we provided. Indeed, if you combine ideas from HW2 with a delete function, you will have the basic pieces needed to solve this problem.

Ignore knights and pawns for now. Say you want to check if the piece at (x, y) can moves to (x', y'). First, find out what piece is at (x, y). (This is already solved in HW2.) Using the attack method, you can determine if the piece can move to (x', y'). Now for the first technical part. If the piece can move to (x', y'), you would need to output the actual path from (x, y) to (x', y'). You can use arrays for this, if it makes life easier.

For each square on the path, find if there is another piece on that square. (This can be done using the find method or even the validity checking from HW2.) If so, this blocks the path, so the move is not possible. If all intermediate squares are empty and (x', y') is empty, the move is possible. If (x', y') has a piece of a different color, the move is also possible (and is a capture).

To actually make the move, you need to update the position of the piece. Furthermore, if there was a piece at (x', y'), you need to delete it from the list. The validity checking of HW2 is a great tool for debugging your code.

Knights do not need any path checking, and pawns have different moves depending on whether they attack or not.

Once you have all of this, determining check is not difficult. All you need to do is determine if any Black piece can move to the square with the White king (or vice versa).

2 Detailed instructions

Format: You should provide a Makefile. On running `make`, it should create "ChessMoves.jar" that takes two command line arguments: an input file and an output file. Thus, on running "java -jar ChessMoves.jar input.txt output.txt", it will read the input from "input.txt" and write out the output in "output.txt".

The input file has the following format. Each line represents a new board. It begins with a chessboard, given by a sequence of "char column row", where char is one of k (king), q (queen), r (rook), b (bishop), n (knight), p (pawn).

If the character is capitalized, it denotes black pieces, otherwise, the piece is white. (This is the same as in HW2.)

Then, there is a colon (':'). This is the end of the board. What follows the colon is a sequence of moves.

For example, a line could look like:

k 4 4 r 8 2 B 1 1 K 4 7: 8 2 2 2 1 1 3 3

The series of moves is: move piece at (8,2) to (2,2), then the piece at (1,1) to (3,3). The first move is possible, but the second move is not (since the rook will then block the bishop's move).

This pattern of lines continues throughout the input file.

Do not worry about error handling on the input, so you can assume that inputs will always have this format. No piece will be placed outside the chessboard, and each square will have at most one piece. Every input board will have exactly one king of each color, just like regular chess.

Output: On running `java -jar ChessMoves.jar input.txt output.txt`, a file "output.txt" should be produced. Each line of the output file corresponds to a line of the input file. Each line will looks like either of the following:

- "Legal": this simply means that the sequence of moves was legal.
- "<MOVE> illegal": This is the output if one of the moves is illegal, as described in the section earlier. <MOVE> lists the illegal move, as 4 integers with spaces between them, indicating the source column, row and the destination column, row (just like the input file). The move is illegal because one of the conditions of a legal move fails.

  For our example above, the output should be:

  1 1 3 3 illegal

Examples: The website contains a zip file called `Examples.zip`. In this, there are numerous example input and output files. The checker will use `simple-input.txt` and `simple-output.txt`. Being such a nice person, I have also put the jar file for my own solution. Among other things, it prints the board to the console after every single move, and gives an explanation for any illegal move encountered. Hopefully, this will aid you in building more test cases.

3 Grading

You code should terminate within 1 minute per every line of input. Pawns are only for extra credit. For all other settings in the rubric below, it does not have to work with pawns.

1. (11 points, extra credit) A full solution that works with pawns.
2. (10 points) A full solution
3. (8 points) Does not deal with detecting check, but works otherwise.

4. (6 points) Does not deal with detecting blocks, but works otherwise.