CMPS 12B-01, Fall 2018: HW 2
Maintaining your chessboard

• 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. Assume an 8×8 chessboard. Implement the following procedures that, given a chessboard:

• Determine validity: verify that two pieces do not occupy the same square.

• Find piece: given a square, determine (if any) the chess piece at that square.

• Determine attack (from that piece): determine if the piece found above attacks another piece. Note that the pieces must be of opposite colors. Do not worry about blocking, so assume that pieces can "pass through" other pieces on the board. This is not true for proper chess, but makes life much easier for this assignment. Thus, a piece x is said to attack another pieces y, if x can travel to y's square (according to the rules of chess) assuming there are no other pieces on the board.

• Do not worry about non-standard moves such as en passant or castling.

You will not get any credit if you do not implement using a linked list. Furthermore, you have to write your own linked list from scratch. You cannot use built in libraries for linked lists.

Suggestions for coding: You do not have to follow these instructions, but it might make the code nicer. Create a (super)class ChessPiece with children subclasses for each different type of piece. Each of these can implement their own attacking function. Think about where you want to store the position and color of each piece.

Create a class ChessBoard that has a linked list of ChessPiece objects. This class can have methods for determining validity, finding, etc.

Format: You should provide a Makefile. On running `make`, it should create "Chessboard.jar". It takes two command line arguments: an input file and an output file

The format of the input file is as follows. The line starts with two integers, referring to a chessboard. This is followed by a colon. Then, the line contains a list of pieces denoted by "char column row", where char is one of k (king), q (queen), r (rook), b (bishop), n (knight). No pawns for now. If the character is capitalized, it denotes black pieces, otherwise, the piece is white. The next line simply contains the coordinates of a single square. For example, these could look like:

8 2: q 4 3 k 4 4 r 8 2 R 8 8 b 1 1 K 4 8 N 7 7

The portion after the colon denotes a chessboard, and the portion before the colon is a query asking "what is at (8,2)". The answer here is a white rook, denoted r. This pattern of lines continues throughout input.txt.

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. (You may want to use the "split" method in java for splitting strings.)

Output: On running (say) `java -jar Chessboard.jar in.txt output.txt`, the input file `in.txt` is read in, and the output file `out.txt` should be produced. The ith line of the output file corresponds to the ith chessboard. The ith line of the output file should contain:

- If the ith chessboard is not valid, the line should be "Invalid".
- If the ith chessboard is valid: first output the chesspiece at the query square. If none, print "-". If there is piece at the query square, print a space, and then "y" (the piece attacks some other piece) or "n" (it does not attack another piece). Pay attention to the space. For the example given above, the output is:

  r y

Examples: The website contains a zip file called `HW2-examples.zip`. The files `test-input.txt` and `test-output.txt` are for the checker. There are more tests in `more-input.txt` and `more-output.txt`.

Helper code: The `PrintSolutionSnippet.java` file contains the `printSolution()` method. It can be used to print out your solution on the chessboard. You pass it a 2D $s \times s$ character array that contains your solution. Compile it and run it to see some examples. Check out the documentation to see how to use the methods. You will probably want to integrate the methods into your code to print your boards. You can copy this code into your code, and use it as you please.

2 Grading

You code should terminate within 5 minutes for all runs. If it doesn't, we will not give you credit.

1. (12 points, extra credit) If you can handle pawns as well, you get extra credit. Pawns will be specified by p (white) or P (black). Note that black and white pawns behave differently, so you need some care in coding this up. White pawns only move "upwards" (increasing row) and black pawns only move "downwards" (decreasing row).

2. (10 points) For a full solution as described above.

3. (8 points) Works without knights.

4. (6 points) Works only when the board has queens.