CMPS 12B-02, Fall 2017
HW2: Managing a Chessboard

Due: Oct 29, 2017 by 11:59pm

- All assignments must be submitted through git. Please look at the Piazza guide on submitting assignments.
- Please follow instructions properly, include naming conventions, and carefully read through the input/output formats. Please review the Piazza guide on the checking script. Run the checking script to make sure your files are named correctly. You will get no credit if the checking script fails!
- 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 Instructor or the TAs.

1 Problem description

Main objective: Store the chess pieces on an 8 x 8 chessboard as a linked list. Implement the following procedures for a given chessboard:

- **Determine Validity**: Verify that two pieces do not occupy the same square, and there is exactly one king of each color (white and black).
- **Find Piece** on square: Given a square, determine the chess piece at that square, if any. As in HW1, squares are specified as (column, row).
- **Discover an Attack** (by a particular piece): Find out if the piece (found by Find Piece) attacks another piece. Note that the pieces must be of different colors. Do not worry about blocking--assume that pieces can "pass through" other pieces on the board. This is not true for proper chess, but it makes coding much easier for this assignment. Thus a piece x on a square can attack another piece y on a different square if x would be able travel to y's square (according to the rules of chess) if there were no other pieces on the board. (You'll just have to find one attack, not all of them.)
- Do not work about non-standard moves, such as castling.

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

Suggestions for coding: You do not have to follow the following instructions, but they might make life easier for you (and your code nicer). Create a (super)class Chess Piece with child subclasses for each different type of piece. Each of these subclasses can implement its own attacking function. Think about how you want to represent the position and color of each piece.

Create a class ChessBoard that has a Linked List of ChessPiece objects. (You must do this via a Linked List.) This class can have methods for determining validity, finding, etc.

Format: You should provide a Makefile. Running `make` should create "ChessBoard.jar". Your program should take two command line arguments, an input file and an output file.

The format of the input file is as follows. Each line starts with two integers, referring to a specific chessboard position; the integers specify a column and a row. This is followed by a colon. Then the line contains a list of pieces consisting of "piece column row", where piece is a character that is one of k (king), q (queen), r (rook), b (bishop). (There are no pawns or knights in HW2.) If the character is capitalized, then the piece is black; if it is not capitalized, then the piece is white. For example, a line could be:

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

The portion after the colon describes the pieces on an 8 x 8 chessboard, and the portion before the colon is a query asking "what is at position (8,2)". The answer here is a white rook, denoted r. This pattern of pairs of lines continues throughout the input file.

Do not worry about error handling on the input; you can assume that inputs will always have this format. No piece will be placed outside the chessboard. If you want to, you can use the Java "split" method to split strings; there's a good example of its use with white space (such as blanks) here.

Output: On running (say) :

```
java -jar ChessBoard.jar in.txt out.txt
```

the file `in.txt` is read in and the output file `out.txt` should be produced. The i'th line of the output file `out.txt` corresponds to the i'th chessboard in `in.txt`. The i'th line of the output file should contain:

- If the i'th chessboard is not valid, then the i'th line should be "Invalid" (without the quotes).
- If the i'th chessboard is valid, then first output the chess piece (if any) that's at the query square. If there is none, then just print "-" (without the quotes). If there is a piece at the query square, print a space, and then print "y" (the piece attacks some other piece) or "n" (it does not attack another piece). For the example given above, the output would be:

  r y

  because the white rook at (8.2) attacks a black rook at (8,8).

Examples: Piazza has a zip file for HW2 called `HW2files.zip`. The files `test-input.txt` and `test-output.txt` (which should help you see if your program is correct) are for the checker. There will be two more files added soon (`more-input.txt` and `more-output.txt`), which will also help you test your program.

Helper code: The `PrintSolutionSnippet.java` file on Piazza which is also in `HW2files.zip` has a `printSolution()` method. That can be used to print out your solution on the chessboard. You pass it a 2 dimensional character array that contains a chessboard. You will probably want to integrate the methods into your code to print your boards. You can copy this code into your own code, and use it as you please. It is provided to help you visualize the chessboard; it is not a required part of HW2.

2 Grading

You code should terminate within 5 minutes for all runs. If it doesn't, we will not give you credit. As usual, your submission must past the checker to get credit; the spec file, HW2-spec.txt, is in the same zip file, `HW2files.zip`, that was mentioned above. To receive credit ,you also must submit your git commit id using the Google Form for HW2.

1. (10 points) Full solution as described above.
2. (9 points) Works for attacks and also for either finding or validity, but not for both.
3. (8 points) Works for attacks, but not for either finding or validity.
4. (7 points) Works for validity and finding, but not for attacks. This is much easier, since you don't even need to code up an attack function.
5. (6 points) Works for finding. Also, the validity check verifies that there is one king of each color, but doesn't check that there is at most one piece on each square.
6. (5 points) Works only for finding.
7. (3 points) The code only verifies that there is one black king and one white king, but does not fully check for validity.