```
import random
import re


def location2index(loc: str) -> tuple[int, int]:
    """converts chess location to corresponding x and y coordinates"""
    col = ord(loc[0]) - 96
    row = int(loc[1:])
    return col, row


def index2location(x: int, y: int) -> str:
    """converts the pair of coordinates to corresponding location"""
    col = chr(x + 96)
    return f'{col}{y}'


class Piece:
    pos_x: int
    pos_y: int
    side: bool  # True for White and False for Black

    def __init__(self, pos_x: int, pos_y: int, side_: bool):
        """sets initial values"""
        self.pos_x = pos_x
        self.pos_y = pos_y
        self.side = side_

    # b is of type Board, but since it is not instantiated until after Pieces,
type hint is not used for any of the
    # below functions

    def can_reach(self, pos_x: int, pos_y: int, b) -> bool:
        """
        checks if this piece can move to coordinate pos_x, pos_y
        on board B according to rule [Rule3] (see section Intro)
        Hint: use is_piece_at
        """

        # Check if location is out of bounds
        if (pos_x > b[0] or pos_y > b[0]) or (pos_x <= 0 or pos_y <= 0):
            return False

        # Rule 3 -- location occupied
        occupied = location_occupied(pos_x, pos_y, b)
        if occupied[0] and occupied[1].side == self.side:
            return False

        return True

    def can_move_to(self, pos_x: int, pos_y: int, b) -> bool:
        """
        checks if this piece can move to coordinate pos_x, pos_y
        on board B according to all chess rules
```

```
        Hints:
        - firstly, check [Rule1] and [Rule3] using can_reach
        - secondly, check if result of move is capture using is_piece_at
        - if yes, find the piece captured using piece_at
        - thirdly, construct new board resulting from move
        - finally, to check [Rule4], use is_check on new board
        """
        init_x, init_y = self.pos_x, self.pos_y
        if self.can_reach(pos_x, pos_y, b):
            occupied = location_occupied(pos_x, pos_y, b)
            if occupied[0] and occupied[1].side == self.side:
                return False
            elif occupied[0] and occupied[1].side != self.side:
                p = piece_at(pos_x, pos_y, b)
                if type(p) == Knight:
                    b[1].remove(p)
                self.pos_x, self.pos_y = pos_x, pos_y
                if is_check(self.side, b):
                    self.pos_x, self.pos_y = init_x, init_y
                    if type(p) == Knight:
                        b[1].append(p)
                    return False
                if type(p) == Knight:
                    b[1].append(p)
            elif not occupied[0]:
                self.pos_x, self.pos_y = pos_x, pos_y
                if is_check(self.side, b):
                    self.pos_x, self.pos_y = init_x, init_y
                    return False
            self.pos_x, self.pos_y = init_x, init_y
            return True
        return False

    def move_to(self, pos_x: int, pos_y: int, b):
        """
        returns new board resulting from move of this piece to coordinates
pos_x, pos_y on board B
        assumes this move is valid according to chess rules
        """
        size = b[0]
        pieces = b[1]

        # Remove piece at occupied space if exists and opposite color
        occupied = location_occupied(pos_x, pos_y, b)
        if occupied[0] and occupied[1].side != self.side:
            for piece in pieces:
                if piece == occupied[1]:
                    pieces.remove(piece)

        # Move piece to new position
        self.pos_x = pos_x
        self.pos_y = pos_y
```

```python
        # Reconstruct Board
        b = (size, pieces)

        return b


Board = tuple[int, list[Piece]]


def is_piece_at(pos_x: int, pos_y: int, b: Board) -> bool:
    """checks if there is piece at coordinates pox_X, pos_y of board B"""
    # Check if outside boundaries
    if (pos_x > b[0] or pos_y > b[0]) or (pos_x <= 0 or pos_y <= 0):
        return False
    pieces = b[1]
    found = False
    # Find if there is a piece at requested position
    for piece in pieces:
        if piece.pos_x == pos_x:
            if piece.pos_y == pos_y:
                found = True
    return found


def piece_at(pos_x: int, pos_y: int, b: Board) -> Piece:
    """
    returns the piece at coordinates pox_X, pos_y of board B
    assumes some piece at coordinates pox_X, pos_y of board B is present
    """
    pieces = b[1]
    # Say which piece is at requested position
    for piece in pieces:
        if piece.pos_x == pos_x:
            if piece.pos_y == pos_y:
                return piece


def separate_pieces(pieces: list) -> tuple:
    """separates the board pieces to a list of kings and list of knights"""
    true_knights = []
    true_king = None
    false_knights = []
    false_king = None
    for piece in pieces:
        if type(piece) == Knight:
            if piece.side:
                true_knights.append(piece)
            else:
                false_knights.append(piece)
        else:
            if piece.side:
                true_king = piece
            else:
                false_king = piece
```

```
    return true_king, true_knights, false_king, false_knights


def location_occupied(pos_x: int, pos_y: int, b) -> tuple[bool, Piece]:
    # Rule 3 -- location occupied
    # tuple[0] is if occupied, tuple[1] is piece at position
    occupied = False
    piece = None
    if is_piece_at(pos_x, pos_y, b):
        occupied = True
        p = piece_at(pos_x, pos_y, b)
        piece = p

    return occupied, piece


class Knight(Piece):
    def __init__(self, pos_x: int, pos_y: int, side_: bool):
        """sets initial values by calling the constructor of Piece"""
        super().__init__(pos_x, pos_y, side_)

    def can_reach(self, pos_x: int, pos_y: int, b: Board) -> bool:
        """
        checks if this rook can move to coordinate pos_x, pos_y
        on board B according to rule [Rule1] and [Rule3] (see section Intro)
        Hint: use is_piece_at
        """
        # Get result from superclass
        reach = super().can_reach(pos_x, pos_y, b)

        if reach:
            # Rule 1 -- over 2, up 1 or over 1, up 2
            if pos_x > (self.pos_x + 2) or pos_x < (self.pos_x - 2):
                return False
            if pos_y > (self.pos_y + 2) or pos_y < (self.pos_y - 2):
                return False

            delta_x = abs(self.pos_x - pos_x)
            delta_y = abs(self.pos_y - pos_y)
            if (delta_x == 2 and delta_y != 1) or (delta_y == 2 and delta_x !=
1):
                return False
            if delta_x <= 1 and delta_y <= 1:
                return False

            return True
        return False


class King(Piece):
    def __init__(self, pos_x: int, pos_y: int, side_: bool):
        """sets initial values by calling the constructor of Piece"""
        super().__init__(pos_x, pos_y, side_)
```

```
    def can_reach(self, pos_x: int, pos_y: int, b: Board) -> bool:
        """checks if this king can move to coordinate pos_x, pos_y on board B
according to rule [Rule2] and [Rule3]"""
        # Get result from superclass
        reach = super().can_reach(pos_x, pos_y, b)

        if reach:
            # Rule 2 -- any direction by one square
            if pos_x > (self.pos_x + 1) or pos_x < (self.pos_x - 1):
                return False
            if pos_y > (self.pos_y + 1) or pos_y < (self.pos_y - 1):
                return False

            delta_x = abs(self.pos_x - pos_x)
            delta_y = abs(self.pos_y - pos_y)
            if (delta_x == 1 and delta_y > 1) or (delta_y == 1 and delta_x >
1):
                return False

            return True
        else:
            return False


def possible_king_move(king: King, b: Board) -> tuple[bool, list[tuple[int,
int]]]:
    proposed_moves = [(king.pos_x + 1, king.pos_y),
                      (king.pos_x + 1, king.pos_y + 1),
                      (king.pos_x, king.pos_y + 1),
                      (king.pos_x - 1, king.pos_y + 1),
                      (king.pos_x - 1, king.pos_y),
                      (king.pos_x - 1, king.pos_y - 1),
                      (king.pos_x, king.pos_y - 1),
                      (king.pos_x + 1, king.pos_y - 1)]

    can_move = False
    possible_moves = []

    for move in proposed_moves:
        if king.can_move_to(move[0], move[1], b):
            can_move = True
            possible_moves.append(move)

    return can_move, possible_moves


def possible_knight_move(knight: Knight, b: Board) -> tuple[bool,
list[tuple[int, int]]]:
    proposed_moves = [(knight.pos_x + 2, knight.pos_y + 1),
                      (knight.pos_x + 2, knight.pos_y - 1),
                      (knight.pos_x - 2, knight.pos_y + 1),
                      (knight.pos_x - 2, knight.pos_y - 1),
                      (knight.pos_x + 1, knight.pos_y + 2),
                      (knight.pos_x - 1, knight.pos_y + 2),
```

```
                      (knight.pos_x + 1, knight.pos_y - 2),
                      (knight.pos_x - 1, knight.pos_y - 2)]

    can_move = False
    possible_moves = []

    for move in proposed_moves:
        if knight.can_move_to(move[0], move[1], b):
            can_move = True
            possible_moves.append(move)

    return can_move, possible_moves


def is_check(side: bool, b: Board) -> bool:
    """
    checks if configuration of B is checked for side
    Hint: use can_reach
    """
    # White is True, Black is False
    # separated order: true_kings, true_knights, false_kings, false_knights
    pieces = b[1]
    separated = separate_pieces(pieces)
    if side:
        knights = separated[3]
        safe_king = separated[2]
        troubled_king = separated[0]
    else:
        knights = separated[1]
        safe_king = separated[0]
        troubled_king = separated[2]

    for knight in knights:
        if knight.can_reach(troubled_king.pos_x, troubled_king.pos_y, b):
            return True
    if safe_king.can_reach(troubled_king.pos_x, troubled_king.pos_y, b):
        return True
    return False


def is_checkmate(side: bool, b: Board) -> bool:
    """
    checks if configuration of B is checkmate for side

    Hints:
    - use is_check
    - use can_reach
    """
    # White is True, Black is False
    # separated order: true_kings, true_knights, false_kings, false_knights
    pieces = b[1]
    separated = separate_pieces(pieces)
    if side:
        troubled_king = separated[0]
```

```
        knights = separated[1]
        opposite_pieces = [separated[2]] + separated[3]
    else:
        troubled_king = separated[2]
        knights = separated[3]
        opposite_pieces = [separated[0]] + separated[1]

    # Piece putting king in check
    check_piece = None
    for oppose in opposite_pieces:
        if oppose.can_reach(troubled_king.pos_x, troubled_king.pos_y, b):
            check_piece = oppose

    # Check if king can move
    troubled = possible_king_move(troubled_king, b)

    # If king can move
    if troubled[0] or not is_check(side, b):
        return False
    # If check and king cannot move
    if is_check(side, b) and not troubled[0]:
        # If knights can take out Check piece
        for knight in knights:
            if knight.can_reach(check_piece.pos_x, check_piece.pos_y, b):
                return False
    return True


def is_stalemate(side: bool, b: Board) -> bool:
    """
    checks if configuration of B is stalemate for side

    Hints:
    - use is_check
    - use can_move_to
    """
    # White is True, Black is False
    # separated order: true_kings, true_knights, false_kings, false_knights
    pieces = b[1]
    separated = separate_pieces(pieces)
    if side:
        king = separated[0]
        knights = separated[1]
    else:
        king = separated[2]
        knights = separated[3]

    # Check if king has a move
    king_moves = possible_king_move(king, b)
    # Check if knights have a move
    knights_can_move = False
    for knight in knights:
        knights_move = possible_knight_move(knight, b)
        if knights_move[0]:
```

```
            knights_can_move = True
            break

    # If check, not a stalemate
    if is_check(side, b):
        return False
    # If any piece can move, not a stalemate
    if king_moves[0] or knights_can_move:
        return False
    return True


def split_move(move: str) -> tuple:
    """splits up the user move input"""
    # Check move string for split between current position to move position
    second_col_index = 2
    for x, char in enumerate(move):
        if char.isdigit():
            continue
        second_col_index = x

    current_pos_str = move[:second_col_index]
    proposed_pos_str = move[second_col_index:]

    return current_pos_str, proposed_pos_str


def is_valid_move(move: str, b: Board) -> bool:
    """checks if move is valid on board"""
    size = b[0]

    moves = split_move(move)

    # Convert string to integers
    current_pos = location2index(moves[0])
    proposed_pos = location2index(moves[1])

    # Check the bounds of the move
    if (current_pos[0] > size or current_pos[0] <= 0) or (current_pos[1] > size
or current_pos[1] <= 0):
        return False
    if (proposed_pos[0] > size or proposed_pos[0] <= 0) or (proposed_pos[1] >
size or proposed_pos[1] <= 0):
        return False

    # Check if there is a piece at current position
    if not is_piece_at(current_pos[0], current_pos[1], b):
        return False

    # Check if piece at current can get to proposed
    piece = piece_at(current_pos[0], current_pos[1], b)
    if not piece.can_reach(proposed_pos[0], proposed_pos[1], b):
        return False
```

```
    return True


def from_file_to_piece(p: str, side: bool, size: int) -> Piece:
    which = p[0]
    loc = p[1:]
    location = location2index(loc)

    if which not in ['N', 'K']:
        raise IOError

    if location[0] > size or location[1] > size:
        raise IOError

    if which == 'N':
        return Knight(location[0], location[1], side)
    else:
        return King(location[0], location[1], side)


def read_board(filename: str) -> Board:
    """
    reads board configuration from file in current directory in plain format
    raises IOError exception if file is not valid (see section Plain board
configurations)
    """
    # Check if file exists
    try:
        fhand = open(filename)
    except FileNotFoundError:
        raise IOError

    # Split the lines into a list then close file
    file_lines = [line.strip() for line in fhand]
    fhand.close()

    # Check if the first line is a digit for board size
    if not file_lines[0].isdigit():
        raise IOError
    size = int(file_lines[0])

    if size < 3 or size > 26:
        raise IOError

    # Split pieces lines into lists
    white_pieces_str = re.split(' *, *', file_lines[1])
    black_pieces_str = re.split(' *, *', file_lines[2])

    # Get ready to check for the pieces
    white_pieces = []
    black_pieces = []
    # Add white pieces
    for piece in white_pieces_str:
        white_pieces.append(from_file_to_piece(piece, True, size))
```

```
    # Add black pieces
    for piece in black_pieces_str:
        black_pieces.append(from_file_to_piece(piece, False, size))

    # Get number of pieces
    num_of_white_pieces = len(white_pieces)
    num_of_black_pieces = len(black_pieces)

    # Check if more than one king
    num_of_white_kings = 0
    num_of_black_kings = 0
    for piece in white_pieces:
        if type(piece) == King:
            num_of_white_kings += 1
    for piece in black_pieces:
        if type(piece) == King:
            num_of_black_kings += 1

    if (num_of_white_kings > 1 or num_of_white_kings <= 0) or
(num_of_black_kings > 1 or num_of_black_kings <= 0):
        raise IOError

    # Check the number of knights
    num_of_white_knights = num_of_white_pieces - num_of_white_kings
    num_of_black_knights = num_of_black_pieces - num_of_black_kings
    total_num_of_knights = num_of_black_knights + num_of_white_knights

    if total_num_of_knights > (size ** 2) - 2:
        raise IOError

    # Combine all the pieces into one list
    pieces = white_pieces + black_pieces

    return size, pieces


def save_board(filename: str, b: Board) -> None:
    """saves board configuration into file in current directory in plain
format"""
    b_size = b[0]
    pieces = b[1]
    black_str = ''
    white_str = ''

    # Get string format of index
    for x, piece in enumerate(pieces):
        if type(piece) == King:
            piece_str = f'K{index2location(piece.pos_x, piece.pos_y)}'
        else:
            piece_str = f'N{index2location(piece.pos_x, piece.pos_y)}'

        # Check the color of the piece
        if piece.side:
            white_str += piece_str
```

```
            if x < (len(pieces) - 1):
                white_str += ', '
        else:
            black_str += piece_str
            if x < (len(pieces) - 1):
                black_str += ', '

    # Search for extension
    ext = filename.find('.')
    if ext == -1:
        filename += '.txt'

    # Open file and save
    with open(filename, 'w+') as fname:
        fname.write(str(b_size))
        fname.write(white_str)
        fname.write(black_str)


def can_capture(side: bool, b: Board) -> tuple[bool, Piece, int, int]:
    """checks if a piece can be captured"""
    can = False  # Bool if can_capture
    cannot = True
    capture = None  # Piece that does the capture
    x = 0  # X position for Piece to move to
    y = 0  # Y position for Piece to move to

    # Get pieces
    separated = separate_pieces(b[1])
    if side:
        knights = separated[1]
        king = separated[0]
    else:
        knights = separated[3]
        king = separated[2]

    # Get king moves
    king_moves = possible_king_move(king, b)

    # Find a capture
    while True:
        # Check if king can capture a piece
        if king_moves[0]:
            for move in king_moves[1]:
                if is_piece_at(move[0], move[1], b):
                    piece = piece_at(move[0], move[1], b)
                    if piece.side != side:
                        x = move[0]
                        y = move[1]
                        capture = king
                        can = True
                        break

        # Check if knight can capture a piece
```

```
        for knight in knights:
            found = False
            knight_moves = possible_knight_move(knight, b)
            for move in knight_moves[1]:
                if is_piece_at(move[0], move[1], b):
                    piece = piece_at(move[0], move[1], b)
                    if piece.side != side:
                        x = move[0]
                        y = move[1]
                        capture = knight
                        can = True
                        cannot = False
                        found = True
                        break
            if found:
                break
        if can or cannot:
            break

    return can, capture, x, y


def find_black_move(b: Board) -> tuple[Piece, int, int]:
    """
    returns (P, x, y) where a Black piece P can move on B to coordinates x,y
according to chess rules
    assumes there is at least one black piece that can move somewhere

    Hints:
    - use methods of random library
    - use can_move_to
    """
    size = b[0]
    pieces = separate_pieces(b[1])
    black_king = pieces[2]
    black_knights = pieces[3]
    all_blacks = black_knights + [black_king]
    can_cap = can_capture(False, b)
    can_move_pieces = []
    moved = False
    x = 0
    y = 0

    # Check if king is in check
    if is_check(False, b):
        p = black_king

        # If king can capture safely
        if can_cap[0] and type(can_cap[1]) == King:
            if black_king.can_move_to(can_cap[2], can_cap[3], b):
                x = can_cap[2]
                y = can_cap[3]
            # Move King out of the way
            else:
```

```
                while not moved:
                    pos_x = random.randint(black_king.pos_x - 1,
black_king.pos_x + 1)
                    pos_y = random.randint(black_king.pos_y - 1,
black_king.pos_y + 1)
                    if (pos_x > size or pos_y > size) or (pos_x <= 0 or pos_y
<= 0):
                        continue
                    if black_king.can_move_to(pos_x, pos_y, b):
                        x = pos_x
                        y = pos_y
                        moved = True
        # Move King out of the way
        else:
            while not moved:
                pos_x = random.randint(black_king.pos_x - 1, black_king.pos_x +
1)
                pos_y = random.randint(black_king.pos_y - 1, black_king.pos_y +
1)
                if (pos_x > size or pos_y > size) or (pos_x <= 0 or pos_y <=
0):
                    continue
                if black_king.can_move_to(pos_x, pos_y, b):
                    x = pos_x
                    y = pos_y
                    moved = True
    elif can_cap[0]:
        p = can_cap[1]
        x = can_cap[2]
        y = can_cap[3]
    else:
        # Move any piece that is able to
        while not moved:
            if not black_knights:
                pos_x = random.randint(black_king.pos_x - 1, black_king.pos_x +
1)
                pos_y = random.randint(black_king.pos_y - 1, black_king.pos_y +
1)
                if (pos_x > size or pos_y > size) or (pos_x <= 0 or pos_y <=
0):
                    continue
            # pick a random move
            else:
                pos_x = random.randint(0, size)
                pos_y = random.randint(0, size)
            for black in all_blacks:
                if black.can_move_to(pos_x, pos_y, b):
                    can_move_pieces.append(black)
            if can_move_pieces:
                x = pos_x
                y = pos_y
                moved = True
        p = random.choice(can_move_pieces)
```

```
    return p, x, y


def conf2unicode(b: Board) -> str:
    """converts board configuration B to unicode format string (see section
Unicode board configurations)"""
    # Create matrix of all blank spaces per board size
    brd_matrx = [['\u2001' for _ in range(b[0])] for _ in range(b[0])]
    pieces = b[1]
    brd_str = ''
    # Create unicode of each piece
    for piece in pieces:
        if piece.side and (type(piece) == King):
            peace = '\u2654'
        elif piece.side and (type(piece) == Knight):
            peace = '\u2658'
        elif not piece.side and (type(piece) == King):
            peace = '\u265A'
        else:
            peace = '\u265E'

        # Get position of the piece
        col = piece.pos_x - 1
        row = piece.pos_y - 1

        # Put correct unicode into position
        brd_matrx[row][col] = peace

    # Convert list to a string to show as board
    for i in range(len(brd_matrx)):
        for j in brd_matrx[-(i+1)]:
            brd_str += j
        brd_str += '\n'

    return brd_str


def main() -> None:
    """
    runs the play

    Hint: implementation of this could start as follows:
    filename = input("File name for initial configuration: ")
    ...
    """

    fname = input('File name for initial configuration: ')
    while True:
        if fname.upper() == 'QUIT':
            print('Quitting program')
            quit()
        try:
            board = read_board(fname)
            break
```

```
        except IOError:
            fname = input('This is not a valid file. File name for initial
configuration: ')

    print('The initial configuration is:')
    print(conf2unicode(board))

    if is_checkmate(True, board):
        pass

    move = input('Next move of White: ')
    while move.upper() != 'QUIT':
        if not is_valid_move(move, board):
            move = input('This is not a valid move. Next move of White: ')
            continue

        moves = split_move(move)
        # Convert string to integers
        current_pos = location2index(moves[0])
        proposed_pos = location2index(moves[1])

        piece = piece_at(current_pos[0], current_pos[1], board)
        board = piece.move_to(proposed_pos[0], proposed_pos[1], board)

        print("The configuration after White's move is:")
        print(conf2unicode(board))

        if is_checkmate(False, board):
            print('Game over. White wins.')
            quit()
        if is_stalemate(False, board):
            print('Game over. Stalemate.')
            quit()
        if is_check(False, board):
            print('White Check Black.')

        black_to_move = find_black_move(board)
        black_move = index2location(black_to_move[1], black_to_move[2])
        black_init_pos = index2location(black_to_move[0].pos_x,
black_to_move[0].pos_y)
        board = black_to_move[0].move_to(black_to_move[1], black_to_move[2],
board)

        print(f"Next move of Black is {black_init_pos}{black_move}. The
configuration after Black's move is:")
        print(conf2unicode(board))

        if is_checkmate(True, board):
            print('Game over. Black wins.')
            quit()
        if is_stalemate(True, board):
            print('Game over. Stalemate.')
            quit()
        if is_check(True, board):
```

```
            print('Black Check White.')

        move = input('Next move of White: ')

    save_name = input('File name to store the configuration: ')
    save_board(save_name, board)
    print('The game configuration saved.')


if __name__ == '__main__':  # keep this in
    main()
```