**Chess Engine**

**in JavaScript**

****

**­­­­**

* Ashish Papanai

**Day-1**

**Understanding the basics of chess.**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| A8 | B8 | C8 | D8 | E8 | F8 | G8 | H8 |
| A7 | B7 | C7 | D7 | E7 | F7 | G7 | H7 |
| A6 | B6 | C6 | D6 | E6 | F6 | G6 | H6 |
| A5 | B5 | C5 | D5 | E5 | F5 | G5 | H5 |
| A4 | B4 | C4 | D4 | E4 | F4 | G4 | H4 |
| A3 | B3 | C3 | D3 | E3 | F3 | G3 | H3 |
| A2 | B2 | C2 | D2 | E2 | F2 | G2 | H2 |
| A1 | B1 | C1 | D1 | E1 | F1 | G1 | H1 |

**Rank**

**Files**

Fig (a): This is a representation of an actual chess board with each Horizontal Row represents a rank.

Representation followed in this engine.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| A1 | B1 | C1 | D1 | E1 | F1 | G1 | H1 |
| A2 | B2 | C2 | D2 | E2 | F2 | G2 | H2 |
| A3 | B3 | C3 | D3 | E3 | F3 | G3 | H3 |
| A4 | B4 | C4 | D4 | E4 | F4 | G4 | H4 |
| A5 | B5 | C5 | D5 | E5 | F5 | G5 | H5 |
| A6 | B6 | C6 | D6 | E6 | F6 | G6 | H6 |
| A7 | B7 | C7 | D7 | E7 | F7 | G7 | H7 |
| A8 | B8 | C8 | D8 | E8 | F8 | G8 | H8 |

**Rank**

**Files**

Fig (b): This is a representation of a chess board according to the planned code.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |
| 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |

**Rank**

**Files**

Fig (c): Actual Chess Board in a form of 2d matrix

Modified Board representation to prevent the piece from escaping the board

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 |
| 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 |
| 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 |
| 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 |
| 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 |
| 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 |
| 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 |
| 110 | 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 |

Fig 1: Indexing the modified board

If any piece reaches the yellow square, We will not allow and register that move.

Reason for adding layer of two squares on top and bottom:

Rook moves in a L shape in horizontal and vertical direction to forbade the horse from moving out of board in vertical direction we add two yellow boxes, this condition is not possible for horizontal motion of Knight, thus we have only one empty row in the left and right side to prevent.

Similarly, to prevent Rook, Bishop and Queen, we have added One layer at vertical and horizontal positions

**Day-2**

Representation used in the code:

let PIECES = {EMPTY: 0, wP: 1, wN: 2, wB: 3, wR: 4, wQ: 5, wK: 6,

              bP: 7, bN: 8, bR: 10, bQ: 11, bK: 12 };

let BRD\_SQ\_NUM = 120;

let FILES = {FILE\_A: 0, FILE\_B: 1,FILE\_C: 2,FILE\_D: 3,

    FILE\_E: 4,FILE\_F: 5,FILE\_G: 6,FILE\_H: 7, FILE\_NONE: 8};

let RANKS = {RANK\_1: 0, RANK\_2: 1,RANK\_3: 2,RANK\_4: 3,

        RANK\_5: 4,RANK\_6: 5,RANK\_7: 6,RANK\_8: 7, RANK\_NONE: 8};

var COLOURS = {WHITE: 0, BLACK: 1, BOTH:2 };

var SQUARES = {

    A1:21, B1:22, C1:23, D1:24, E1:25, F1:26, G1:27, H1:28,

    A8:91, B8:92, C8:93, D8:94, E8:95, F8:96, G8:97, H8:98,

    NO\_SQ: 99, OFFBOARD: 100

};

wP: White Pawn, wN: White Knight, wB: White Bishop, wR: White Rook, wQ: White Queen, wK: White King

bP: Black Pawn, bN: Black Knight, bB: Black Bishop, bR: Black Rook, bQ: Black Queen, bK: Black King

File representation of the board: x

Fig 2: file representation of the board

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| 100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 100 |
| 100 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 100 |
| 100 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 100 |
| 100 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 100 |
| 100 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 100 |
| 100 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 100 |
| 100 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 100 |
| 100 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 100 |
| 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |

This board represents the file according to the representation used by us in this code.

Implement conversion of files to square:

Refer fig 1 for visualization.

Initialise the board as depicted in fig 2:

function InitFilesRanksBrd(){

    let index = 0;

    let file = FILES.FILE\_A;

    var rank = RANKS.RANK\_1;

    var sq = SQUARES.A1;

    for(index = 0; index < BRD\_SQ\_NUM; ++index){

        FilesBrd[index] = SQUARES.OFFBOARD;

        RanksBrd[index] = SQUARES.OFFBOARD;

    }

    for(rank = RANKS.RANK\_1; rank <= RANKS.RANK\_8; ++rank){

        for(file = FILES.FILE\_A; file <= FILES.FILE\_H; ++file){

            sq = FR2SQ(file, rank);

            FilesBrd[sq] = file;

            RanksBrd[sq] = rank;

        }

    }

}

Castling:

let CASTLEBIT = { WKCA: 1, WQCA: 2, BKCA: 4, BQCA: 8};

White King side Castle: 0001

White Queen side Castle: 0010

Black King side Castle: 0100

Black Queen side Castle: 1000

GameBoard.castlePerm = 0; //will keep a track of castling permission.

If we want to check that castling is possible at that particular side, we’ll take bitwise AND of the corresponding values of the castling sides with the castlePerm, this will tell if that castle is allowed or not.

Snippet to keep a track of all pieces and the piece values

var PieceBig = [ BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.TRUE, BOOL.TRUE, BOOL.TRUE, BOOL.TRUE, BOOL.FALSE, BOOL.TRUE, BOOL.TRUE, BOOL.TRUE, BOOL.TRUE, BOOL.TRUE ];

var PieceMaj = [ BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.TRUE, BOOL.TRUE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.TRUE, BOOL.TRUE ];

var PieceMin = [ BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.TRUE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.TRUE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE ];

var PieceVal= [ 0, 100, 325, 325, 550, 1000, 50000, 100, 325, 325, 550, 1000, 50000  ]; // Value of Piece

var PieceCol = [ COLOURS.BOTH, COLOURS.WHITE, COLOURS.WHITE, COLOURS.WHITE, COLOURS.WHITE, COLOURS.WHITE, COLOURS.WHITE,

    COLOURS.BLACK, COLOURS.BLACK, COLOURS.BLACK, COLOURS.BLACK, COLOURS.BLACK, COLOURS.BLACK ];

var PiecePawn = [ BOOL.FALSE, BOOL.TRUE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE ];

var PieceKnight = [ BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE ];

var PieceKing = [ BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.TRUE ];

var PieceRookQueen = [ BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.TRUE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.TRUE, BOOL.FALSE ];

var PieceBishopQueen = [ BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.FALSE, BOOL.TRUE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.FALSE, BOOL.TRUE, BOOL.FALSE ];

var PieceSlides = [ BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.TRUE, BOOL.TRUE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.TRUE, BOOL.TRUE, BOOL.FALSE ];

Explanation of the variables:

* PieceBig: Tells if a piece is a pawn or a non-pawn piece.
* PieceMaj: Stores if a piece is a major piece, i.e. Rook and Queen
* PieceMin: Stores if a piece is a minor piece or not, i.e. Bishop and Knight
* PieceVal: stores the value of each piece

For the values of each pieces we keep pawn as a reference, Points of all pieces:

* Pawn - 100
* Knight - 325
* Bishop - 325
* Rook - 550
* Queen - 1000
* King – 50000

PieceCol: Stores the colour of Pieces.

All the elements inside the arrays are indexed according to the key value pairs mentioned in the references to variables in (page no: 5).

Note: We can have at-most 10 pieces of same type.

Algorithm for movement of pieces-

If piece in the square is of the side to move, then we’ll generate moves for that piece.

How to get index?

e.g.

wPNum or wNNum is nothing but the index of number of pieces from the list GameBoard.pceNum.

i.e. if a board has 2 white knights, then

Get location of squares containing that piece-

For(pceNum = 0; pceNum < GameBoard.pceNum[wP]; ++pceNum){

Sq = PlistArray[wP \* 10 + pceNum];

}

wP can be replaced with the pieces we want to consider and find moves of.

**Day-3**

Generating random numbers from 0 to 31 to this will be used later to get a unique piece code.

function RAND\_32() { // This function will generate random numbers from 0 to 32(excluded)

    return math.floor(math.random() \* 32) + 1;

    /\* Alternate: ( Math.floor(Math.random()\*255)+1 << 23) | (Math.floor(Math.random()\*255)+1 << 16) |

                  Math.floor(Math.random()\*255)+1 << 8 | Math.floor(Math.random()\*255)+1  )

    \*/

}

Also added a variable to store if en-passant is allowed to the pawn.

GameBoard.enPas = 0; //for en-Passent

We will create a random key which will store the associated pieces according to their presence in the board,

Algorithm followed:

this will store the randomly generated number.

If the piece is removed from the key, we will take XOR with that piece again and the contribution of that piece to the key will be removed.

Arrays to store the keys

let PieceKeys = new Array(14 \* 120); // for all the pieces

let SideKey; //It's either black or white

let CastleKeys = new Array(16); // because we are using 4 bits to represent castle

PieceKey will store the key related to any piece at any particular position.

SideKey will store the key related to the side i.e. either black or white.

Castlekey will store the castling key wrt to the castling permission which is represented by 4 bits, i.e. 1000, 0100, 0010, 0001.

The following function will generate the position keys based on the algorithm discussed above.

function GeneratePosKey() {

    let sq = 0;

    let finalKey = 0;

    let piece = PIECES.EMPTY;

    for (sq = 0; sq < BRD\_SQ\_NUM; ++sq) {

        piece = GameBoard.pieces[sq];

        if (piece != PIECES.EMPTY && piece != SQUARES.OFFBOARD) {

            finalKey ^= PieceKeys[(piece \* 120) + sq];

        }

    }

    if (GameBoard.side == COLOURS.WHITE) {

        finalKey ^= SideKey;

    }

    if (GameBoard.enPas != SQUARES.NO\_SQ) {

        finalKey ^= PieceKeys[GameBoard.enPas];

    }

    finalKey ^= CastleKeys[GameBoard.castlePerm];

}

FinalKey: Stores the final key,

piece: stores the current piece on the square,

sq: iterates through all squares of the board.

We will have to switch quite often between the two representations of a chess engine (fig (c), and fig (2)).

To accomplish this, two new functions are created:

The function definitions are self-explanatory and as follows:

function InitSq120To64() {

    let index = 0;

    let file = FILES.FILE\_A;

    let rank = RANKS.RANK\_A;

    let sq = SQUARES.A1;

    let sq64 = 0;

    for (index = 0; index < BRD\_SQ\_NUM; ++index) {

        Sq120ToSq64[index] = 65;

    }

    for (index = 0; index < 64; ++index) {

        Sq64ToSq120[index] = 120;

    }

    for (rank = RANKS.RANK\_1; rank <= RANKS.RANK\_8; ++rank) {

        for (file = FILES.FILE\_A; file <= FILES.FILE\_H; ++file) {

            sq = FR2SQ(file, rank);

            Sq64ToSq120[sq64] = sq;

            Sq120ToSq64[sq] = sq64;

            sq64++;

        }

    }

}

Utility arrays to keep track of moves and move scores.

GameBoard.moveList = new Array(MAXDEPTH \* MAXPOSITIONMOVES);

GameBoard.moveScores = new Array(MAXDEPTH \* MAXPOSITIONMOVES);

GameBoard.moveListStart = new Array(MAXDEPTH);

Added FEN (Forsyth-Edward Notation) Refer [Wikipedia](https://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation) for more details.

To implement FEN, we will have to reset the board, the following function will handle the task of resetting the board for us.

function ResetBoard() {

    let index = 0;

    for (index = 0; index < BRD\_SQ\_NUM; ++index) {

        GameBoard.pieces[index] = SQUARES.OFFBOARD;

    }

    for (index = 0; index < 64; ++index) {

        GameBoard.pieces[SQ120(index)] = PIECES.EMPTY;

    }

    for (index = 0; index < 14 \* 120; ++index) {

        GameBoard.pList[index] = PIECES.EMPTY;

    }

    for (index = 0; index < 2; ++index) {

        GameBoard.material[index] = 0;

    }

    for (index = 0; index < 13; ++index) {

        GameBoard.pceNum[index] = 0;

    }

    GameBoard.side = COLOURS.BOTH;

    GameBoard.enPas = SQUARES.NO\_SQ;

    GameBoard.fiftyMove = 0;

    GameBoard.ply = 0;

    GameBoard.hisPly = 0;

    GameBoard.CastlePerm = 0;

    GameBoard.posKey = 0;

    GameBoard.moveListStart[GameBoard.ply] = 0;

}

Explanation:  
all the pieces are set off board with the first loop, the second board sets all the pieces to empty pieces in 120 bases indexes, third loop removes all pieces in the piece list, fourth loop removes all the captured materials and the last loop will reset the number of pieces.

All other lines are self-explanatory.

function ParseFen(fen) {

    ResetBoard();

    let rank = RANKS.RANK\_0;

    let file = FILES.FILE\_A;

    let piece = 0;

    let count = 0;

    let i = 0;

    let sq120 = 0;

    let fenCnt = 0;

    while ((rank >= RANKS.RANK\_1) && fenCnt < fen.length) {

        count = 1;

        switch (fen[fenCnt]) {

            case 'p':

                piece = PIECES.bP;

                break;

            case 'r':

                piece = PIECES.bR;

                break;

            case 'n':

                piece = PIECES.bN;

                break;

            case 'b':

                piece = PIECES.bB;

                break;

            case 'k':

                piece = PIECES.bK;

                break;

            case 'q':

                piece = PIECES.bQ;

                break;

            case 'P':

                piece = PIECES.wP;

                break;

            case 'R':

                piece = PIECES.wR;

                break;

            case 'N':

                piece = PIECES.wN;

                break;

            case 'B':

                piece = PIECES.wB;

                break;

            case 'K':

                piece = PIECES.wK;

                break;

            case 'Q':

                piece = PIECES.wQ;

                break;

            case '1':

            case '2':

            case '3':

            case '4':

            case '5':

            case '6':

            case '7':

            case '8':

                piece = PIECES.EMPTY;

                count = fen[fenCnt].charCodeAt() - '0'.charCodeAt();

                break;

            case '/':

            case ' ':

                rank--;

                file = FILES.FILE\_A;

                fenCnt++;

                continue;

            default:

                console.log("FEN error");

                return;

        }

        for (i = 0; i < count; i++) {

            sq120 = FR2SQ(file, rank);

            GameBoard.pieces[sq120] = piece;

            file++;

        }

        fenCnt++;

    }

}

The switch case condition of the FEN parser will check if the character at particular index is a number or an alphabet, an Alphabet would represent a piece and a number would represent empty squares.

GameBoard.side = (fen[fenCnt] == 'w') ? COLOURS.WHITE : COLOURS.BLACK;

    fenCnt += 2;

The code mentioned above will handle which side should move in the FEN notation.

for (i = 0; i < 4; i++) {

        if (fen[fenCnt] == ' ') {

            break;

        }

        switch (fen[fenCnt]) {

            case 'K':

                GameBoard.castlePerm |= CASTLEBIT.WKCA;

                break;

            case 'Q':

                GameBoard.castlePerm |= CASTLEBIT.WQCA;

                break;

            case 'k':

                GameBoard.castlePerm |= CASTLEBIT.BKCA;

                break;

            case 'q':

                GameBoard.castlePerm |= CASTLEBIT.BQCA;

                break;

            default:

                break;

        }

        fenCnt++;

    }

    fenCnt++;

The code mentioned above will handle the castling permission according to the notations discussed before.

The following code will handle the conditions for en-passant and after setting the necessary conditions, we will generate the position keys accordingly.

   if (fen[fenCnt] != '-') {

        file = fen[fenCnt].charCodeAt() - 'a'.charCodeAt();

        rank = fen[fenCnt + 1].charCodeAt() - '1'.charCodeAt();

        console.log("fen[fenCnt]:" + fen[fenCnt] + " File:" + file + " Rank:" + rank);

        GameBoard.enPas = FR2SQ(file, rank);

    }

    GameBoard.posKey = GeneratePosKey();