

Chess Engine

in JavaScript

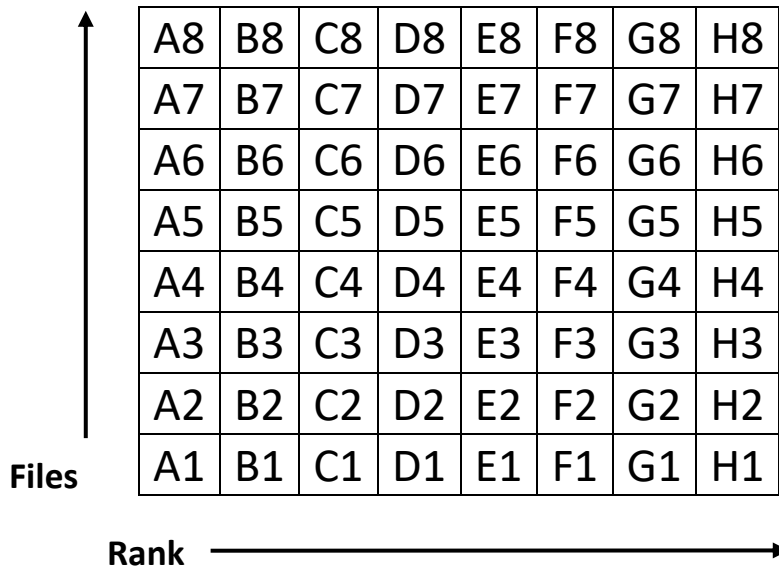


- Ashish Papanai

-

Day-1

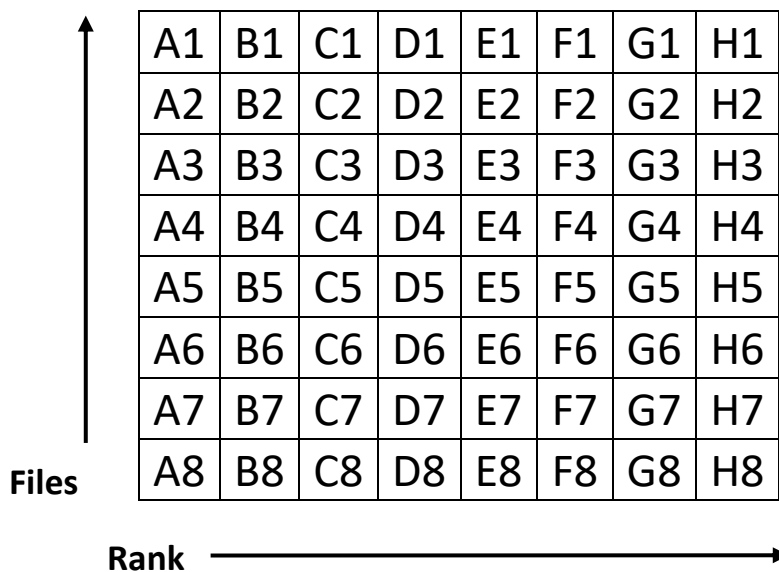
Understanding the basics of chess.



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

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

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

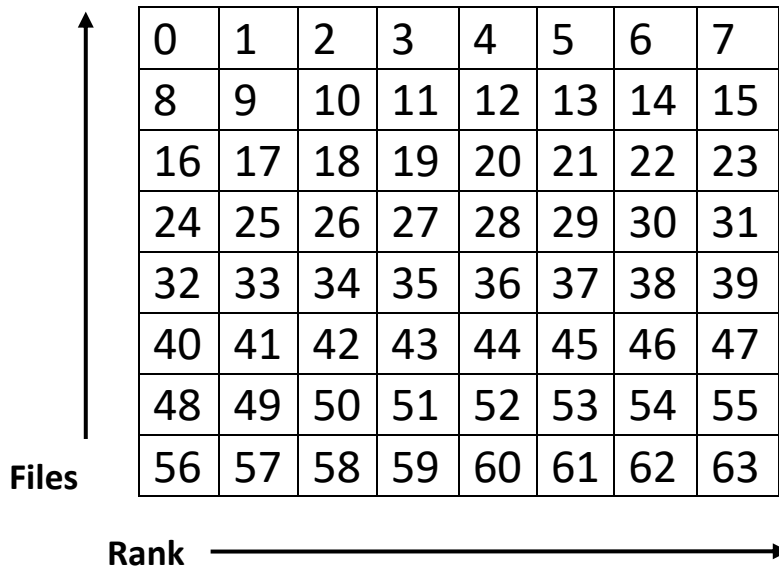


Fig (c): Actual Chess Board in a form of 2d matrix

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

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

Fig 2: file representation of the board

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

Implement conversion of files to square:

$$(file + 21) + (rank \times 10)$$

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

var PiecePawn = [ BOOL.FALSE, BOOL.TRUE, 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.FALSE, BOOL.FALSE ];
var PieceKnight = [ BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE ];
var PieceKing = [ BOOL.FALSE, 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.FALSE, BOOL.TRUE, BOOL.TRUE ];
var PieceRookQueen = [ BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, BOOL.TRUE, BOOL.TRUE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.FALSE, BOOL.TRUE, 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.FALSE, BOOL.TRUE, 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.FALSE, BOOL.TRUE, BOOL.TRUE, BOOL.TRUE, BOOL.FALSE, BOOL.FALSE, 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.

$$sqOfpiece = pieceListArray[index]$$

How to get index?

$$\text{e.g. } wP \text{ index} = wP * 10 + wPNum$$

$$wN \text{ index} = wN * 10 + wNNum$$

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

$$GameBoard.pceNum[wP] = 2$$

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(a)

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:

let piece = RAND_32()

this will store the randomly generated number.

let key = 0

key = key XOR piece

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:

sq64to120() and sq120to64()

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-Edwards_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();
```

Making the Update FEN textbox interactive by JQuery:

```
$("#SetFen").click(function () {
    var fenStr = $("#fenIn").val();
    ParseFen(fenStr);
    PrintBoard();
});
```

This CSS snippet will increase the size of text box and will improve its positioning.

```
#FenInDiv {
    position: relative;
    left: 60px;
}

#fenIn {
    width: 480px;
    left: 60px;
}
```

After changing the FEN notation, we need to update the piece list according to the pieces set on the board,

```
function UpdateListMaterial(){
    var piece, sq, index, colour;
```

```

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;
}

for(index = 0; index < 64; ++index){
    sq = SQ120(index);
    piece = GameBoard.pieces[sq];
    if(piece != PIECES.EMPTY){
        colour = PieceCol[piece];

        GameBoard.material[colour] += PieceVal[piece];

        GameBoard.pList[PCEINDEX(piece,GameBoard.pceNum[piece])] = sq;
        GameBoard.pceNum[piece]++;
    }
}
PrintPieceLists();
}

```

The function mentioned above will handle all the functionalities of handling the updating piece list after updating the FEN Notation.

Day-3(b)

Check if a square is under attack:

Condition of attack follow the chess rules, Pawn attacks diagonally, Rook attacks in a full horizontal or full vertical direction, bishop attacks diagonally, Rook attacks in L horizontal or L vertical, Queen attacks as a combination of Rook and Bishop and King can attack one square in all directions.

1. Square under attack due to a pawn:

```

if(side == COLOURS.WHITE){
    if(GameBoard.pieces[sq - 11] == PIECES.wP || GameBoard.pieces[sq - 9]
    == PIECES.wP ){

```

```

        return BOOL.TRUE;
    }
}
else{
    if(GameBoard.pieces[sq + 11] == PIECES.bP || GameBoard.pieces[sq + 9]
== PIECES.bP ){
        return BOOL.TRUE;
    }
}
}

```

2. Square under attack by Knight:

```

for(index = 0; index < 8; index++) {
    pce = GameBoard.pieces[sq + KnDir[index]];
    if(pce != SQUARES.OFFBOARD && PieceCol[pce] == side && PieceKnight[pce]
] == BOOL.TRUE) {
        return BOOL.TRUE;
    }
}
}

```

3. Square under attack by rook:

```

for(index = 0; index < 4; ++index) {
    dir = RkDir[index];
    t_sq = sq + dir;
    pce = GameBoard.pieces[t_sq];
    while(pce != SQUARES.OFFBOARD) {
        if(pce != PIECES.EMPTY) {
            if(PieceRookQueen[pce] == BOOL.TRUE && PieceCol[pce] == side)
{
                return BOOL.TRUE;
            }
            break;
        }
        t_sq += dir;
        pce = GameBoard.pieces[t_sq];
    }
}
}

```

4. Square under attack by bishop:

```

for(index = 0; index < 4; ++index) {
    dir = BiDir[index];
    t_sq = sq + dir;

```

```

        pce = GameBoard.pieces[t_sq];
        while(pce != SQUARES.OFFBOARD) {
            if(pce != PIECES.EMPTY) {
                if(PieceBishopQueen[pce] == BOOL.TRUE && PieceCol[pce] == side
) {
                    return BOOL.TRUE;
                }
                break;
            }
            t_sq += dir;
            pce = GameBoard.pieces[t_sq];
        }
    }
}

```

5. Square under attack by King:

```

    for(index = 0; index < 8; index++) {
        pce = GameBoard.pieces[sq + KiDir[index]];
        if(pce != SQUARES.OFFBOARD && PieceCol[pce] == side && PieceKing[pce]
== BOOL.TRUE) {
            return BOOL.TRUE;
        }
    }
}

```

Directions of all major pieces are tracked with the help of arrays whose definitions are:

```

let KnDir = [ -8, -19, -21, -12, 8, 19, 21, 12 ];
let RkDir = [ -1, -10, 1, 10 ];
let BiDir = [ -9, -11, 11, 9 ];
let KiDir = [ -1, -10, 1, 10, -9, -11, 11, 9 ];

```

Day-3(c)

For implementing the movement of pieces, we will follow the following steps:

1. Source square will be obtained by calculating binary AND of the square and 0xF
2. Destination square will be obtained by right shift of bits by 7

3. If a piece is captured, we'll take binary right shift with 14 / 0xF
4. En-passant will be considered by taking AND 0x40000, i.e. ending the binary representation with 4 trailing groups with zeros and the fifth group as 0100.
5. Pawn start will be represented by taking AND with 0x80000, i.e. ending the binary representation with 4 groups of trailing zeros and the fifth group as 1000.
6. A promoted piece will be considered by right shifting by 20, 0xF
0000 1111 0000 0000 0000 0000
7. Castling will be represented by taking AND with 0x1000000
0001 0000 0000 0000 0000 0000 0000

```
function FROMSQ(m) { return (m & 0x7F); }
function TOSQ(m) { return ( (m >> 7) & 0x7F); }
function CAPTURED(m) { return ( (m >> 14) & 0xF); }
function PROMOTED(m) { return ( (m >> 20) & 0xF); }

let MFLAGEP = 0x40000;
let MFLAGPS = 0x80000;
let MFLAGCA = 0x100000;

let MFLAGCAP = 0x7C000;
let MFLAGPROM = 0xF00000;

let NOMOVE = 0;
```

Generating moves for the pawns: the pawn can move straight, takes diagonally, can promote itself and has a special movement of en-passant.

```
for(pceNum = 0; pceNum < GameBoard.pceNum[pceType]; ++pceType) {
    sq = GameBoard.pList[PCEINDEX(pceType, pceNum)];

    if(GameBoard.pieces[sq + 10] == PIECES.EMPTY) {
        // Add Pawn Move Here
        if(RanksBrd[sq] == RANKS.RANK_2 && GameBoard.pieces[sq + 20] =
= PIECES.EMPTY) {
            // Add Quiet Move Here
```

```

    }
}

    if(SQOFFBOARD(sq + 9) == BOOL.FALSE && PieceCol[GameBoard.pieces[sq+9]] == COLOURS.BLACK) {
        // Add Pawn Cap Move
    }

    if(SQOFFBOARD(sq + 11) == BOOL.FALSE && PieceCol[GameBoard.pieces[sq+11]] == COLOURS.BLACK) {
        // Add Pawn Cap Move
    }

    if(GameBoard.enPas != SQUARES.NOSQ) {
        if(sq + 9 == GameBoard.enPas) {
            // Add enPas Move
        }

        if(sq + 11 == GameBoard.enPas) {
            // Add enPas Move
        }
    }
}

```

This code will work for the white pawns, for black pawns we will change the location/Rank of the pieces.

The following code handles castling

```

if(GameBoard.castlePerm & CASTLEBIT.WKCA) {
    if(GameBoard.pieces[SQUARES.F1] == PIECES.EMPTY && GameBoard.pieces[SQUARES.G1] == PIECES.EMPTY) {
        if(SqAttacked(SQUARES.F1, COLOURS.BLACK) == BOOL.FALSE && SqAttacked(SQUARES.E1, COLOURS.BLACK) == BOOL.FALSE) {
            // Add Quiet Move
        }
    }
}

if(GameBoard.castlePerm & CASTLEBIT.WQCA) {
    if(GameBoard.pieces[SQUARES.D1] == PIECES.EMPTY && GameBoard.pieces[SQUARES.C1] == PIECES.EMPTY && GameBoard.pieces[SQUARES.B1] == PIECES.EMPTY) {
        if(SqAttacked(SQUARES.D1, COLOURS.BLACK) == BOOL.FALSE && SqAttacked(SQUARES.E1, COLOURS.BLACK) == BOOL.FALSE) {
            // Add Quiet Move
        }
    }
}

```

```
}
```

Capture Moves and Quiet Moves:

Move function is to check for the source, destination square, if we have to capture a piece or promote a piece or if we have an associated flag with the motion of the piece.

```
function MOVE(from, to, captured, promoted, flag) {  
    return (from | (to << 7) | (captured << 14) | (promoted << 20) | flag);  
}
```

AddCaptureMove is to add capture moves to the move list:

```
function AddCaptureMove(move) {  
    GameBoard.moveList[GameBoard.moveListStart[GameBoard.ply+1]] = move;  
    GameBoard.moveScores[GameBoard.moveListStart[GameBoard.ply+1]++] = 0;  
}
```

AddQuietMove is to add non capturing moves to the move list:

```
function AddQuietMove(move) {  
    GameBoard.moveList[GameBoard.moveListStart[GameBoard.ply+1]] = move;  
    GameBoard.moveScores[GameBoard.moveListStart[GameBoard.ply+1]++] = 0;  
}
```

AddEnpassantMove will add en-passant moves to the move list:

```
function AddEnPassantMove(move) {  
    GameBoard.moveList[GameBoard.moveListStart[GameBoard.ply+1]] = move;  
    GameBoard.moveScores[GameBoard.moveListStart[GameBoard.ply+1]++] = 0;  
}
```

The following function are used to add non capturing moves, We will check two condition:

1. Check if we want to promote the piece to a Queen, Rook, Bishop, Knight.
2. We'll check if we want to move a normal pawn move, i.e. no capture, en-passant or promotions.

```
function AddWhitePawnQuietMove(from, to) {  
    if(RanksBrd[from]==RANKS.RANK_7) {  
        AddQuietMove(MOVE(from,to,PIECES.EMPTY,PIECES.wQ,0));  
        AddQuietMove(MOVE(from,to,PIECES.EMPTY,PIECES.wR,0));  
        AddQuietMove(MOVE(from,to,PIECES.EMPTY,PIECES.wB,0));  
    }
```

```

        AddQuietMove(MOVE(from,to,PIECES.EMPTY,PIECES.wN,0));
    } else {
        AddQuietMove(MOVE(from,to,PIECES.EMPTY,PIECES.EMPTY,0));
    }
}

function AddBlackPawnQuietMove(from, to) {
    if(RanksBrd[from]==RANKS.RANK_2) {
        AddQuietMove(MOVE(from,to,PIECES.EMPTY,PIECES.bQ,0));
        AddQuietMove(MOVE(from,to,PIECES.EMPTY,PIECES.bR,0));
        AddQuietMove(MOVE(from,to,PIECES.EMPTY,PIECES.bB,0));
        AddQuietMove(MOVE(from,to,PIECES.EMPTY,PIECES.bN,0));
    } else {
        AddQuietMove(MOVE(from,to,PIECES.EMPTY,PIECES.EMPTY,0));
    }
}

```

The following function are used to add capturing moves, we will check two condition:

1. Check if we want to promote the piece to a Queen, Rook, Bishop, Knight after capturing the opponent's piece.
2. We'll check if we want to capture piece without any promotions.

```

function AddWhitePawnCaptureMove(from, to, cap) {
    if(RanksBrd[from]==RANKS.RANK_7) {
        AddCaptureMove(MOVE(from, to, cap, PIECES.wQ, 0));
        AddCaptureMove(MOVE(from, to, cap, PIECES.wR, 0));
        AddCaptureMove(MOVE(from, to, cap, PIECES.wB, 0));
        AddCaptureMove(MOVE(from, to, cap, PIECES.wN, 0));
    } else {
        AddCaptureMove(MOVE(from, to, cap, PIECES.EMPTY, 0));
    }
}

function AddBlackPawnCaptureMove(from, to, cap) {
    if(RanksBrd[from]==RANKS.RANK_2) {
        AddCaptureMove(MOVE(from, to, cap, PIECES.bQ, 0));
        AddCaptureMove(MOVE(from, to, cap, PIECES.bR, 0));
        AddCaptureMove(MOVE(from, to, cap, PIECES.bB, 0));
        AddCaptureMove(MOVE(from, to, cap, PIECES.bN, 0));
    } else {
        AddCaptureMove(MOVE(from, to, cap, PIECES.EMPTY, 0));
    }
}

```



```

}
}

```

The following segment of code is used for non-sliding pieces like Knight and King:

```

pceIndex = LoopNonSlideIndex[GameBoard.side];
pce = LoopNonSlidePce[pceIndex++];

while (pce != 0) {
    for(pceNum = 0; pceNum < GameBoard.pceNum[pce]; ++pceNum) {
        sq = GameBoard.pList[PCEINDEX(pce, pceNum)];

        for(index = 0; index < DirNum[pce]; index++) {
            dir = PceDir[pce][index];
            t_sq = sq + dir;

            if(SQOFFBOARD(t_sq) == BOOL.TRUE) {
                continue;
            }

            if(GameBoard.pieces[t_sq] != PIECES.EMPTY) {
                if(PieceCol[GameBoard.pieces[t_sq]] != GameBoard.side) {
                    AddCaptureMove( MOVE(sq, t_sq, GameBoard.pieces[t_sq],
PIECES.EMPTY, 0 ));
                }
                else {
                    AddQuietMove( MOVE(sq, t_sq, PIECES.EMPTY, PIECES.EMPTY, 0
));
                }
            }
        }
        pce = LoopNonSlidePce[pceIndex++];
    }
}

```

We will follow a similar approach for sliding pieces like bishop, rook and queen:

```

pceIndex = LoopSlideIndex[GameBoard.side];
pce = LoopSlidePce[pceIndex++];

while(pce != 0) {
    for(pceNum = 0; pceNum < GameBoard.pceNum[pce]; ++pceNum) {
        sq = GameBoard.pList[PCEINDEX(pce, pceNum)];

```

```

        for(index = 0; index < DirNum[pce]; index++) {
            dir = PceDir[pce][index];
            t_sq = sq + dir;

            while( SQOFFBOARD(t_sq) == BOOL.FALSE ) {

                if(GameBoard.pieces[t_sq] != PIECES.EMPTY) {
                    if(PieceCol[GameBoard.pieces[t_sq]] != GameBoard.side)
                {
                    AddCaptureMove( MOVE(sq, t_sq, GameBoard.pieces[t_
sq], PIECES.EMPTY, 0 ));
                }
                break;
            }
            AddQuietMove( MOVE(sq, t_sq, PIECES.EMPTY, PIECES.EMPTY, 0
));
            t_sq += dir;
        }
    }
    pce = LoopSlidePce[pceIndex++];
}

```

Day-4

For defining the make move function, we need to hash the moves to make the function and operations faster:

HASH_PCE: this will calculate the Hash code of the piece by taking XOR of the position key with Piece key at the index of the square on which the piece is present.

HASH_CA: Adds the castle permission to the hash key by taking XOR of the position key with the castling key based on current castling permissions.

HASH_SIDE: Adds which side is to move next to the hash key by taking XOR of the position key with the side key.

HASH_EP: Adds the en-passant permission to the Hash Key by XOR of position key with the en-passant permissions associated with the Piece key.

```
function HASH_PCE(pce, sq) {
  GameBoard.posKey ^= PieceKeys[(pce * 120) + sq];
}

function HASH_CA() {
  GameBoard.posKey ^= CastleKeys[GameBoard.castlePerm];
}

function HASH_SIDE() {
  GameBoard.posKey ^= SideKey;
}

function HASH_EP() {
  GameBoard.posKey ^= PieceKeys[GameBoard.enPas];
}
```

To make a move we will need to clear the location of the current piece, i.e. the piece to be moved, the following function will help us in implementing clear location.

```
function ClearPiece(sq) {

  let pce = GameBoard.pieces[sq];
```

```

    let col = PieceCol[pce];
    let index;
    let t_pceNum = -1;

    HASH_PCE(pce, sq);

    GameBoard.pieces[sq] = PIECES.EMPTY;
    GameBoard.material[col] -= PieceVal[pce];

    for(index = 0; index < GameBoard.pceNum[pce]; ++index) {
        if(GameBoard.pList[PCEINDEX(pce,index)] == sq) {
            t_pceNum = index;
            break;
        }
    }

    GameBoard.pceNum[pce]--;
    GameBoard.pList[PCEINDEX(pce, t_pceNum)] = GameBoard.pList[PCEINDEX(pce, G
ameBoard.pceNum[pce])];
}

```

After removing the piece, we will add the piece to its new location, the following function will fulfil that for us:

```

function AddPiece(sq, pce) {

    let col = PieceCol[pce];

    HASH_PCE(pce, sq);

    GameBoard.pieces[sq] = pce;
    GameBoard.material[col] += PieceVal[pce];
    GameBoard.pList[PCEINDEX(pce, GameBoard.pceNum[pce])] = sq;
    GameBoard.pceNum[pce]++;

}

```

After getting the colour of our piece we will get the hash key of the piece followed by a check for any special moves.

The above mentioned, AddPiece function will be followed by a MovePiece function which will move the piece.

```

function MovePiece(from, to) {

```

```

    let index = 0;
    let pce = GameBoard.pieces[from];

    HASH_PCE(pce, from);
    GameBoard.pieces[from] = PIECES.EMPTY;

    HASH_PCE(pce,to);
    GameBoard.pieces[to] = pce;

    for(index = 0; index < GameBoard.pceNum[pce]; ++index) {
        if(GameBoard.pList[PCEINDEX(pce,index)] == from) {
            GameBoard.pList[PCEINDEX(pce,index)] = to;
            break;
        }
    }
}

```

After the Add and Move piece function, we will create a checkboard function to ensure proper debugging and error handling in the code.

1. Error in Piece list will be handled by the following loop:

```

for(t_piece = PIECES.wP; t_piece <= PIECES.bK; ++t_piece) {
    for(t_pce_num = 0; t_pce_num < GameBoard.pceNum[t_piece]; ++t_pce_num)
    {
        sq120 = GameBoard.pList[PCEINDEX(t_piece,t_pce_num)];
        if(GameBoard.pieces[sq120] != t_piece) {
            console.log('Error Pce Lists');
            return BOOL.FALSE;
        }
    }
}

```

2. Error in the number of function will be handled by:

```

for(t_piece = PIECES.wP; t_piece <= PIECES.bK; ++t_piece) {
    if(t_pceNum[t_piece] != GameBoard.pceNum[t_piece]) {
        console.log('Error t_pceNum');
        return BOOL.FALSE;
    }
}

```

3. All other errors (like, error in material, side to play, position key) will be check by the following conditions:

```
if(t_material[COLOURS.WHITE] != GameBoard.material[COLOURS.WHITE] ||
    t_material[COLOURS.BLACK] != GameBoard.material[COLOURS.BLACK]) {
    console.log('Error t_material');
    return BOOL.FALSE;
}

if(GameBoard.side!=COLOURS.WHITE && GameBoard.side!=COLOURS.BLACK) {
    console.log('Error GameBoard.side');
    return BOOL.FALSE;
}

if(GeneratePosKey()!=GameBoard.posKey) {
    console.log('Error GameBoard.posKey');
    return BOOL.FALSE;
}
```

If no error is found in any part of the code, BOOL.TRUE will be returned.

```
return BOOL.TRUE;
```

Make Move Function:

All details and hash keys collected so far will be used in this function, the conditions of castle, en-passant, fifty moves are easy and self-explanatory.

```
function MakeMove(move) {

    let from = FROMSQ(move);
    let to = TOSQ(move);
    let side = GameBoard.side;

    GameBoard.history[GameBoard.hisPly].posKey = GameBoard.posKey;

    if( (move & MFLAGEP) != 0) {
        if(side == COLOURS.WHITE) {
            ClearPiece(to-10);
        } else {
```

```

        ClearPiece(to+10);
    }
} else if( (move & MFLAGCA) != 0) {
    switch(to) {
        case SQUARES.C1:
            MovePiece(SQUARES.A1, SQUARES.D1);
            break;
        case SQUARES.C8:
            MovePiece(SQUARES.A8, SQUARES.D8);
            break;
        case SQUARES.G1:
            MovePiece(SQUARES.H1, SQUARES.F1);
            break;
        case SQUARES.G8:
            MovePiece(SQUARES.H8, SQUARES.F8);
            break;
        default: break;
    }
}

if(GameBoard.enPas != SQUARES.NO_SQ) HASH_EP();
HASH_CA();

GameBoard.history[GameBoard.hisPly].move = move;
GameBoard.history[GameBoard.hisPly].fiftyMove = GameBoard.fiftyMove;
GameBoard.history[GameBoard.hisPly].enPas = GameBoard.enPas;
GameBoard.history[GameBoard.hisPly].castlePerm = GameBoard.castlePerm;

GameBoard.castlePerm &= CastlePerm[from];
GameBoard.castlePerm &= CastlePerm[to];
GameBoard.enPas = SQUARES.NO_SQ;

HASH_CA();

let captured = CAPTURED(move);
GameBoard.fiftyMove++;

if(captured != PIECES.EMPTY) {
    ClearPiece(to);
    GameBoard.fiftyMove = 0;
}

GameBoard.hisPly++;
GameBoard.ply++;

if(PiecePawn[GameBoard.pieces[from]] == BOOL.TRUE) {
    GameBoard.fiftyMove = 0;
    if( (move & MFLAGPS) != 0) {

```

```

        if(side==COLOURS.WHITE) {
            GameBoard.enPas=from+10;
        } else {
            GameBoard.enPas=from-10;
        }
        HASH_EP();
    }
}

MovePiece(from, to);

let prPce = PROMOTED(move);
if(prPce != PIECES.EMPTY) {
    ClearPiece(to);
    AddPiece(to, prPce);
}

GameBoard.side ^= 1;
HASH_SIDE();

if(SqAttacked(GameBoard.pList[PCEINDEX(Kings[side],0)], GameBoard.side))
{
    TakeMove();
    return BOOL.FALSE;
}

return BOOL.TRUE;
}

```

Take Move function will be used to undo move to analyse a better position while exploring your games with this engine, The function will work opposite to the make move function.

Get the most recent move from the move list, get the to and from square, colour of the piece and side to move. Check for any special conditions while making the move and finally check if any piece was captured in the move. If any piece was captured, remove it from the materials and add it back to the board with the help of AddPiece function.


```

function TakeMove() {

    GameBoard.hisPly--;
    GameBoard.ply--;

    let move = GameBoard.history[GameBoard.hisPly].move;
    let from = FROMSQ(move);
    let to = TOSQ(move);

    if(GameBoard.enPas != SQUARES.NO_SQ) HASH_EP();
    HASH_CA();

    GameBoard.castlePerm = GameBoard.history[GameBoard.hisPly].castlePerm;
    GameBoard.fiftyMove = GameBoard.history[GameBoard.hisPly].fiftyMove;
    GameBoard.enPas = GameBoard.history[GameBoard.hisPly].enPas;

    if(GameBoard.enPas != SQUARES.NO_SQ) HASH_EP();
    HASH_CA();

    GameBoard.side ^= 1;
    HASH_SIDE();

    if( (MFLAGEP & move) != 0) {
        if(GameBoard.side == COLOURS.WHITE) {
            AddPiece(to-10, PIECES.bP);
        } else {
            AddPiece(to+10, PIECES.wP);
        }
    } else if( (MFLAGCA & move) != 0) {
        switch(to) {
            case SQUARES.C1: MovePiece(SQUARES.D1, SQUARES.A1); break;
            case SQUARES.C8: MovePiece(SQUARES.D8, SQUARES.A8); break;
            case SQUARES.G1: MovePiece(SQUARES.F1, SQUARES.H1); break;
            case SQUARES.G8: MovePiece(SQUARES.F8, SQUARES.H8); break;
            default: break;
        }
    }

    MovePiece(to, from);

    let captured = CAPTURED(move);
    if(captured != PIECES.EMPTY) {
        AddPiece(to, captured);
    }
}

```

```

    if(PROMOTED(move) != PIECES.EMPTY) {
        ClearPiece(from);
        AddPiece(from, (PieceCol[PROMOTED(move)] == COLOURS.WHITE ? PIECES.wP
: PIECES.bP));
    }
}

```

Perft: Performance Testing

A debugging function is created which will walk through the move generation tree and will count all the legal moves considering the nodes and depth of the tree. The result from this function is verified with results stored in a testing file, if the answers match, the code is correct. Else, it's time to debug.

```

let perft_leafNodes;

function Perft(depth) {

    if(depth == 0) {
        perft_leafNodes++;
        return;
    }

    GenerateMoves();

    let index;
    let move;

    for(index = GameBoard.moveListStart[GameBoard.ply]; index < GameBoard.move
ListStart[GameBoard.ply + 1]; ++index) {

        move = GameBoard.moveList[index];
        if(MakeMove(move) == BOOL.FALSE) {
            continue;
        }
        Perft(depth-1);
        TakeMove();
    }

    return;
}

```

Perft will ignore mate by rotation and fifty move rules of chess.

More details of perft can be obtained [here](#).

PerftTesting function will be done from the following function by giving the depth of the tree:

```
function PerftTest(depth) {  
  
    PrintBoard();  
    console.log("Starting Test To Depth:" + depth);  
    perft_leafNodes = 0;  
  
    let index;  
    let move;  
    let moveNum = 0;  
    GenerateMoves();  
    for(index = GameBoard.moveListStart[GameBoard.ply]; index < GameBoard.move  
ListStart[GameBoard.ply + 1]; ++index) {  
  
        move = GameBoard.moveList[index];  
        if(MakeMove(move) == BOOL.FALSE) {  
            continue;  
        }  
        moveNum++;  
        let cumnodes = perft_leafNodes;  
        Perft(depth-1);  
        TakeMove();  
        let oldnodes = perft_leafNodes - cumnodes;  
        console.log("move:" + moveNum + " " + PrMove(move) + " " + oldnodes);  
    }  
  
    console.log("Test Complete : " + perft_leafNodes + " leaf nodes visited");  
  
    return;  
}
```

Day-5 & 6(a):

Understanding Min-Max search and Alpha beta pruning.

Min-Max Search: (The Brain of the Code)

It's a backtracking algorithm used to build logic for turn based games, we assume that the opponent will make optimal moves and hold a good competition against us, one player tries to maximise the score from the tree while another player tries to minimise the score to win the game. The algorithm follows a depth first search approach, at the terminal node we compare the values of right and left subtrees to check which subtree comes up with an optimum score to end the game.

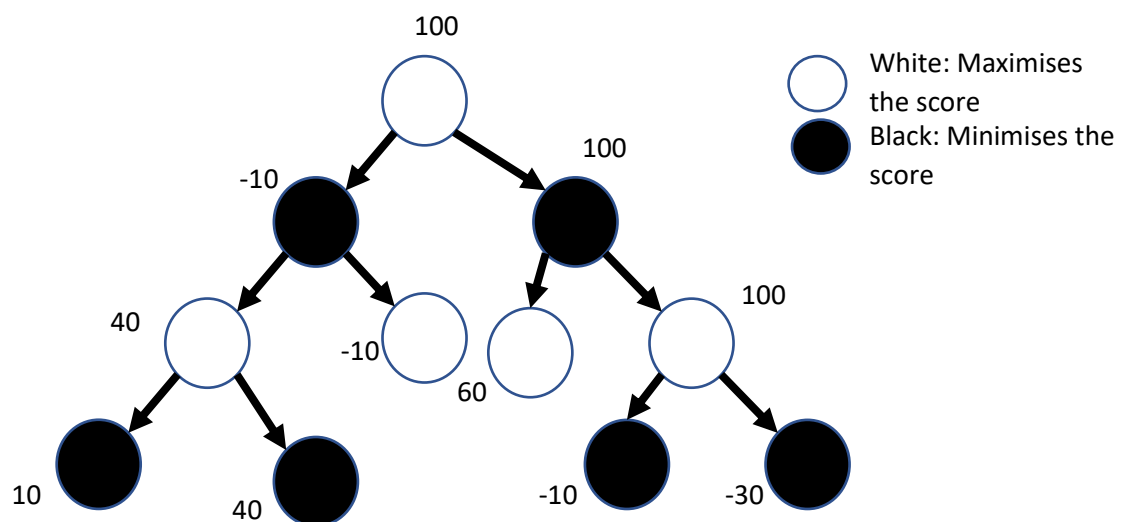


Fig 5: MinMax visualization

This is naïve but a fundamental algorithm for turn based games, an optimisation to this algorithm is the Alpha-Beta Search.

Alpha Beta Pruning: As the name suggests, we will not visit all the leaves to get answer to the problem,

We'll focus on all nodes at same depth and see the minimum/maximum value that can be obtained,

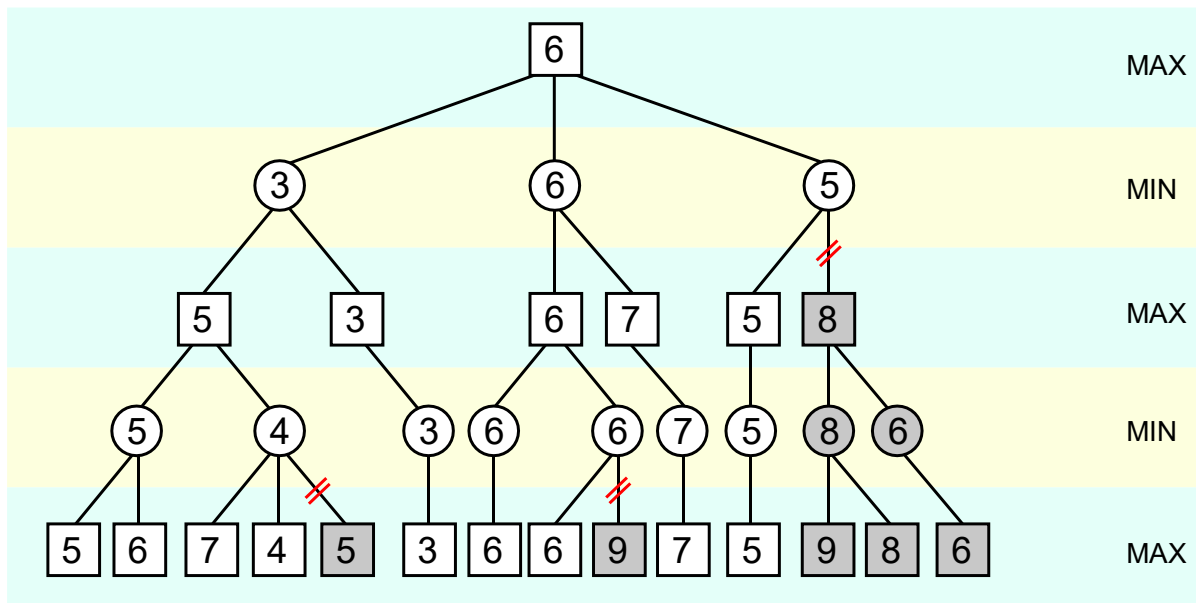


Fig 5: Alpha Beta pruning visualization
(source: [Wikipedia](https://en.wikipedia.org/wiki/Alpha_beta_pruning))

For initial calls, Alpha and Beta will be initialised as ∞ and $-\infty$ respectively.

There's room for improvement in this approach as well and that will be discussed at the end of completion of this project.

The following function will implement Alpha Bets search:

```
function AlphaBeta(alpha, beta, depth) {

    if(depth <= 0) {
        return Quiescence(alpha, beta);
    }

    if ((SearchController.nodes & 2047) == 0) {
        CheckUp();
    }

    SearchController.nodes++;

    if( (IsRepetition() || GameBoard.fiftyMove >= 100) && GameBoard.ply != 0)
    {
        return 0;
    }

    if(GameBoard.ply > MAXDEPTH -1) {
```

```

        return EvalPosition();
    }

    let InCheck = SqAttacked(GameBoard.pList[PCEINDEX(Kings[GameBoard.side],0)
], GameBoard.side^1);
    if(InCheck == BOOL.TRUE) {
        depth++;
    }

    let Score = -INFINITE;

    GenerateMoves();

    let MoveNum = 0;
    let Legal = 0;
    let OldAlpha = alpha;
    let BestMove = NOMOVE;
    let Move = NOMOVE;

    let PvMove = ProbePvTable();
    if(PvMove != NOMOVE) {
        for(MoveNum = GameBoard.moveListStart[GameBoard.ply]; MoveNum < GameBo
ard.moveListStart[GameBoard.ply + 1]; ++MoveNum) {
            if(GameBoard.moveList[MoveNum] == PvMove) {
                GameBoard.moveScores[MoveNum] = 2000000;
                break;
            }
        }
    }

    for(MoveNum = GameBoard.moveListStart[GameBoard.ply]; MoveNum < GameBoard.
moveListStart[GameBoard.ply + 1]; ++MoveNum) {

        PickNextMove(MoveNum);

        Move = GameBoard.moveList[MoveNum];

        if(MakeMove(Move) == BOOL.FALSE) {
            continue;
        }
        Legal++;
        Score = -AlphaBeta( -beta, -alpha, depth-1);

        TakeMove();

        if(SearchController.stop == BOOL.TRUE) {
            return 0;
        }
    }

```

```

        if(Score > alpha) {
            if(Score >= beta) {
                if(Legal == 1) {
                    SearchController.fhf++;
                }
                SearchController.fh++;
                if((Move & MFLAGCAP) == 0) {
                    GameBoard.searchKillers[MAXDEPTH + GameBoard.ply] =
                        GameBoard.searchKillers[GameBoard.ply];
                    GameBoard.searchKillers[GameBoard.ply] = Move;
                }
                return beta;
            }
            if((Move & MFLAGCAP) == 0) {
                GameBoard.searchHistory[GameBoard.pieces[FROMSQ(Move)] * BRD_S
Q_NUM + TOSQ(Move)]
                    += depth * depth;
            }
            alpha = Score;
            BestMove = Move;
        }
    }

    if(Legal == 0) {
        if(InCheck == BOOL.TRUE) {
            return -MATE + GameBoard.ply;
        } else {
            return 0;
        }
    }

    if(alpha != OldAlpha) {
        StorePvMove(BestMove);
    }

    return alpha;
}

function ClearForSearch() {

    let index = 0;
    let index2 = 0;

    for(index = 0; index < 14 * BRD_SQ_NUM; ++index) {
        GameBoard.searchHistory[index] = 0;
    }
}

```

```

    for(index = 0; index < 3 * MAXDEPTH; ++index) {
        GameBoard.searchKillers[index] = 0;
    }

    ClearPvTable();
    GameBoard.ply = 0;
    SearchController.nodes = 0;
    SearchController.fh = 0;
    SearchController.fhf = 0;
    SearchController.start = $.now();
    SearchController.stop = BOOL.FALSE;
}

```

After implementing the Alpha-beta search the condition of 50 moves and repetition are checked because these conditions can end the game with Draw.

```

function IsRepetition() {
    let index = 0;

    for(index = GameBoard.hisPly - GameBoard.fiftyMove; index < GameBoard.hisPly - 1; ++index) {
        if(GameBoard.posKey == GameBoard.history[index].posKey) {
            return BOOL.TRUE;
        }
    }

    return BOOL.FALSE;
}

```

After checking for possible draw conditions there was a need for evaluation table of each pieces to understand the import position for any piece in the game, considering the concept of developing the importance of centre position for pawns and developing knights and bishops.

```

let PawnTable = [
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
10 , 10 , 0 , -10 , -10 , 0 , 10 , 10 ,
5 , 0 , 0 , 5 , 5 , 0 , 0 , 5 ,
0 , 0 , 10 , 20 , 20 , 10 , 0 , 0 ,
5 , 5 , 5 , 10 , 10 , 5 , 5 , 5 ,
10 , 10 , 10 , 20 , 20 , 10 , 10 , 10 ,

```



```

20 , 20 , 20 , 30 , 30 , 20 , 20 , 20 ,
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
];

let KnightTable = [
0 , -10 , 0 , 0 , 0 , 0 , -10 , 0 ,
0 , 0 , 0 , 5 , 5 , 0 , 0 , 0 ,
0 , 0 , 10 , 10 , 10 , 10 , 0 , 0 ,
0 , 0 , 10 , 20 , 20 , 10 , 5 , 0 ,
5 , 10 , 15 , 20 , 20 , 15 , 10 , 5 ,
5 , 10 , 10 , 20 , 20 , 10 , 10 , 5 ,
0 , 0 , 5 , 10 , 10 , 5 , 0 , 0 ,
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
];

let BishopTable = [
0 , 0 , -10 , 0 , 0 , -10 , 0 , 0 ,
0 , 0 , 0 , 10 , 10 , 0 , 0 , 0 ,
0 , 0 , 10 , 15 , 15 , 10 , 0 , 0 ,
0 , 10 , 15 , 20 , 20 , 15 , 10 , 0 ,
0 , 10 , 15 , 20 , 20 , 15 , 10 , 0 ,
0 , 0 , 10 , 15 , 15 , 10 , 0 , 0 ,
0 , 0 , 0 , 10 , 10 , 0 , 0 , 0 ,
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
];

let RookTable = [
0 , 0 , 5 , 10 , 10 , 5 , 0 , 0 ,
0 , 0 , 5 , 10 , 10 , 5 , 0 , 0 ,
0 , 0 , 5 , 10 , 10 , 5 , 0 , 0 ,
0 , 0 , 5 , 10 , 10 , 5 , 0 , 0 ,
0 , 0 , 5 , 10 , 10 , 5 , 0 , 0 ,
0 , 0 , 5 , 10 , 10 , 5 , 0 , 0 ,
25 , 25 , 25 , 25 , 25 , 25 , 25 , 25 ,
0 , 0 , 5 , 10 , 10 , 5 , 0 , 0 ,
];

let BishopPair = 40;

```

Understanding the evaluation tables:

1. Pawn: Good position is controlling the centre and the best position is near promotion

2. Knight: Good position is the squares surrounding the circles and the best position are the centre squares
3. Bishops: Same as Knight, good around the centre squares and best when present at the centre. We will provide bonus point if the user has bishop pair.
4. Rook: Rooks are better while guarding the centre files (d and e) and best when present at the second last file with respect to its side.
5. Queen: follows the evaluation table of rooks and bishop.

These evaluation tables were for white pieces to consider the black pieces as well, we will create mirror evaluation tables.

The pvtable.js file will contain necessary methods to evaluate PV Tables (Principle Variations) table to understand the changes in moves to play and evaluate better.

```
function GetPvLine(depth) {  
  
    let move = ProbePvTable();  
    let count = 0;  
  
    while(move != NOMOVE && count < depth) {  
  
        if( MoveExists(move) == BOOL.TRUE) {  
            MakeMove(move);  
            GameBoard.Pletray[count++] = move;  
        } else {  
            break;  
        }  
        move = ProbePvTable();  
    }  
  
    while(GameBoard.ply > 0) {  
        TakeMove();  
    }  
  
    return count;  
}  
  
function ProbePvTable() {  
    let index = GameBoard.posKey % PVENTRIES;
```

```

    if(GameBoard.PvTable[index].posKey == GameBoard.posKey) {
        return GameBoard.PvTable[index].move;
    }

    return NOMOVE;
}

function StorePvMove(move) {
    let index = GameBoard.posKey % PVENTRIES;
    GameBoard.PvTable[index].posKey = GameBoard.posKey;
    GameBoard.PvTable[index].move = move;
}

```

GetPvLine will get Principle variation line if any move exists in the PvTable, the ProbePvTable will check if the GameBoard.Pvtable contains any entries to evaluate on.

The StorePvMove function will store Position key and moves in GameBoard.PvTable.

After analysis of the position of pieces and creation of PV table we need to detect mate in order to win the game.

The following part of search.js help in detecting possible mates:

```

if(Legal == 0) {
    if(InCheck == BOOL.TRUE) {
        return -MATE + GameBoard.ply;
    } else {
        return 0;
    }
}

```

This will return score indicating a possible mate in the current position, by checking for legal moves with current position of pieces in the GameBoard.

Another possible mate is when we an indicative move from the PV Table, if the move list contains a move already present in the PV Table we will consider that as a possible mate/advantage as well.

```

let PvMove = ProbePvTable();
if(PvMove != NOMOVE) {
    for(MoveNum = GameBoard.moveListStart[GameBoard.ply]; MoveNum < GameBoard.moveListStart[GameBoard.ply + 1]; ++MoveNum) {

```

```

        if(GameBoard.moveList[MoveNum] == PvMove) {
            GameBoard.moveScores[MoveNum] = 2000000;
            break;
        }
    }
}

```

We will use amalgamation of two search moves as the iterative search results in blunder to capture any piece, we can not rely on the evaluation table completely.

This function handles iterative search:

```

function SearchPosition() {

    let bestMove = NOMOVE;
    let bestScore = -INFINITE;
    let currentDepth = 0;
    let line;
    let PvNum;
    let c;
    ClearForSearch();

    for( currentDepth = 1; currentDepth <= /*SearchController.depth*/ 6; ++currentDepth) {

        bestScore = AlphaBeta(-INFINITE, INFINITE, currentDepth);

        if(SearchController.stop == BOOL.TRUE) {
            break;
        }

        bestMove = ProbePvTable();
        line = 'D:' + currentDepth + ' Best:' + PrMove(bestMove) + ' Score:' + bestScore +
            ' nodes:' + SearchController.nodes;

        PvNum = GetPvLine(currentDepth);
        line += ' Pv:';
        for( c = 0; c < PvNum; ++c) {
            line += ' ' + PrMove(GameBoard.Pletray[c]);
        }
        if(currentDepth!=1) {
            line += (" Ordering:" + ((SearchController.fhf/SearchController.fh)*100).toFixed(2) + "%");
        }
    }
}

```

```

        console.log(line);

    }

    SearchController.best = bestMove;
    SearchController.thinking = BOOL.FALSE;
}

```

example of a blunder: Capturing rook with the help of queen completely ignoring the presence of opponent queen or attacking bishop.

These blunders can be avoided by Quiescence search algorithm details are available [here](#) the implementation of Quiescence is mentioned below:

```

function Quiescence(alpha, beta) {

    if ((SearchController.nodes & 2047) == 0) {
        CheckUp();
    }

    SearchController.nodes++;

    if( (IsRepetition() || GameBoard.fiftyMove >= 100) && GameBoard.ply != 0)
    {
        return 0;
    }

    if(GameBoard.ply > MAXDEPTH -1) {
        return EvalPosition();
    }

    let Score = EvalPosition();

    if(Score >= beta) {
        return beta;
    }

    if(Score > alpha) {
        alpha = Score;
    }

    GenerateCaptures();
}

```

```

let MoveNum = 0;
let Legal = 0;
let OldAlpha = alpha;
let BestMove = NOMOVE;
let Move = NOMOVE;

for(MoveNum = GameBoard.moveListStart[GameBoard.ply]; MoveNum < GameBoard.
moveListStart[GameBoard.ply + 1]; ++MoveNum) {

    PickNextMove(MoveNum);

    Move = GameBoard.moveList[MoveNum];

    if(MakeMove(Move) == BOOL.FALSE) {
        continue;
    }
    Legal++;
    Score = -Quiescence( -beta, -alpha);

    TakeMove();

    if(SearchController.stop == BOOL.TRUE) {
        return 0;
    }

    if(Score > alpha) {
        if(Score >= beta) {
            if(Legal == 1) {
                SearchController.fhf++;
            }
            SearchController.fh++;
            return beta;
        }
        alpha = Score;
        BestMove = Move;
    }
}

if(alpha != OldAlpha) {
    StorePvMove(BestMove);
}

return alpha;
}

```

MVV and LVA: (Most valuable victim and less valuable attacker):

By adding this functionality to the engine we can sacrifice less valuable attacker for most valuable victim considering all necessary traps by the Quiescence search, more details about this are available in the reference section at the end of this file and [here](#).

Implementation of MVV and LVA:

```
let MvvLvaValue = [ 0, 100, 200, 300, 400, 500, 600, 100, 200, 300, 400, 500, 600 ];
let MvvLvaScores = new Array(14 * 14);

function InitMvvLva() {
    let Attacker;
    let Victim;

    for(Attacker = PIECES.wP; Attacker <= PIECES.bK; ++Attacker) {
        for(Victim = PIECES.wP; Victim <= PIECES.bK; ++Victim) {
            MvvLvaScores[Victim * 14 + Attacker] = MvvLvaValue[Victim] + 6 - (
MvvLvaValue[Attacker]/100);
        }
    }
}
```

The indexes of MvvLvaValue is based on the index same as the index used to specify the pieces in the pieces array.

Ordering is printed in the console after the end of moves predicted by the engine. Ordering is to verify if the engine is able to predict the correct moves in an optimised manner without wasting time in analysis of useless or less important positions.

This marks the end of backend of the chess engine, the content after this will be focusing on a basic GUI based on basic HTML and CSS.

Day-6(b):

Added a simple UI in HTML CS and linked JavaScript and linked backend features to the front-end.