King's Gambit Chess Developer Manual

KGChess v1.0

Team 20:
King's Gambit

Erick Mercado
Tan Huynh
Samuel Briones-Plascencia
Thanh Tran
Larrenz Carino

UC Irvine EECS 22L Spring 2021

Table of Contents

Glossary

Term	Definition
Algorithm	A procedure or formula used for solving a problem. It is based on conducting a sequence of specified actions in which these actions describe how to do something
API	A set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other service.
AI	A simulation of human intelligence in machines that are programmed to think like humans and mimic their actions.
Array	A data structure consisting of a collection of elements, each identified by at least one array index or key.
B	Bishop Chess Piece:The starting squares are c1 and f1 for White's bishops, and c8 and f8 for Black's bishops.
Char	It stores a single character and requires a single byte of memory in almost all compilers.
Data Types	A particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.
Executable	File that contains a program and is capable of being executed and run on a computer
Flowchart	A diagram of the sequence of movements or actions of people or things involved in a complex system or activity
Functions	A group of statements that together perform a task.A function declaration tells the compiler about a function's name, return type, and parameters.
Int	Short for "integer," is a fundamental variable type built into the compiler and used to define numeric variables holding whole numbers.
K	King Chess Piece: The white chess king begins on the e1 square and the black king begins on the e8 square.
Linked List	a linear collection of data elements whose order is not given by their physical placement in memory. Instead, each element points to

	the next. It is a data structure consisting of a collection of nodes which together represent a sequence.
N	Knight Chess Piece: Travels in L sequences
O.S.	Stands for Operating System, which is a system software that manages computer hardware, software resources, and provides common services for computer
P	Pawn Chess Piece: The most numerous piece moves forward one position at a time, attacks diagonally .
Pointer	A variable in which the value is the address of another variable in memory location.
Q	Queen Chess Piece: Can move in all directions.
R	Rook Chess Piece: can move forward, backward or sideways, but cannot move diagonally
SDL	Short for Simple DirectMedia Layer, it is a cross platform software library used to provide a hardware abstraction layer for multimedia hardware components
Structures	Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of the structured control flow constructs of selection and repetition, block structures, and subroutines.
tar.gz	tar is a computer software utility for collecting many files into one archive file, often referred to as a tarball, for distribution or backup purposes.
2-D Array	An array that consists of more than one rows and more than one column. In 2-D array each element is refer by two indexes. Elements stored in these Arrays in the form of matrices.
Void	When used as a function return type, the void keyword specifies that the function does not return a value. When used for a function's parameter list, void specifies that the function takes no parameters. When used in the declaration of a pointer, void specifies that the pointer is "universal."

1. Software Architecture Overview

1.1 Main Data Types and Structures

- **Data Types**
  - int var for user inputs to navigate through menus
- **Structures**
  - **Board Structure (Doubly linked list)**
    - Data of board pieces in a 64 2-D int array
    - User/AI input for that turn in a 5 char array
    - "Next" pointer that points to next board in list
    - "Prev" pointer that points to the previous board in list

1.2 Major Software Components

- **Diagram of Module Hierarchy:**

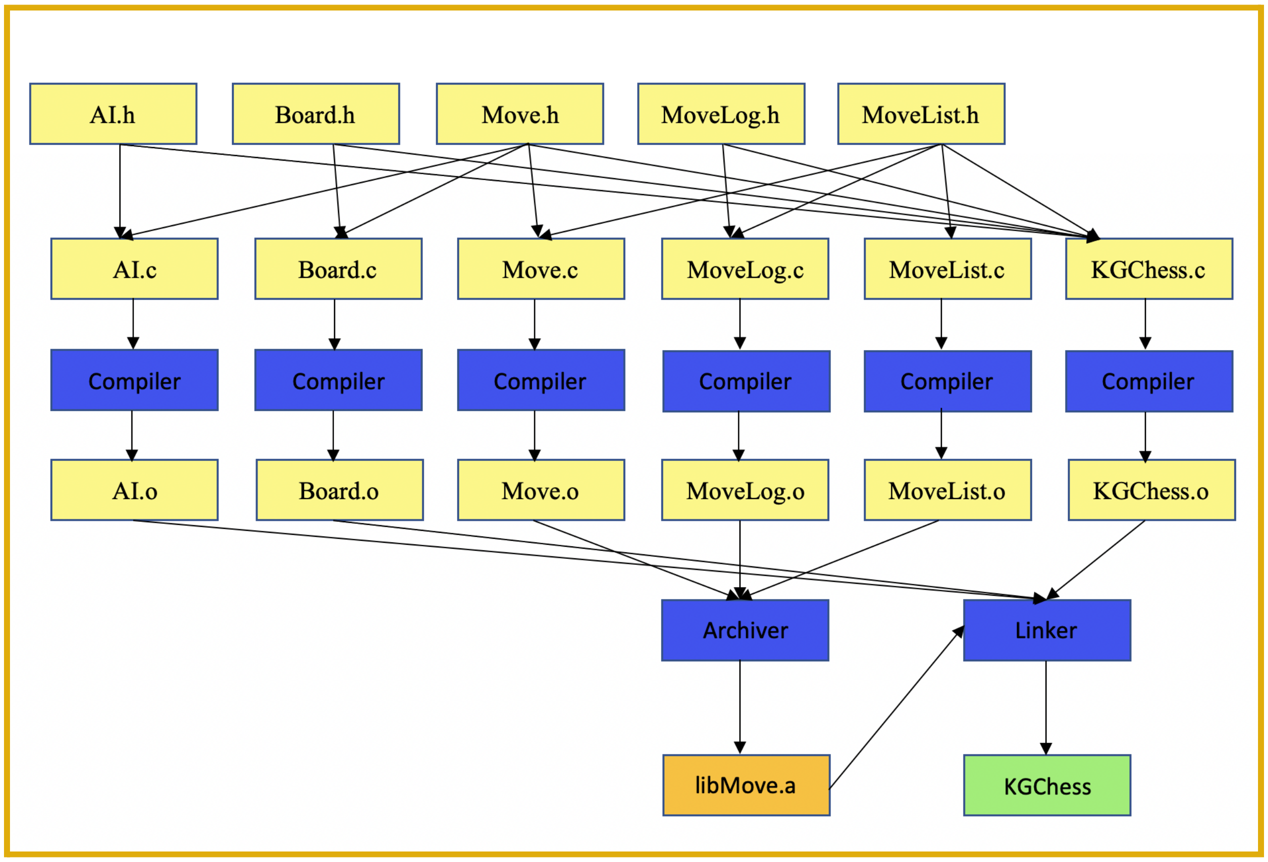


Figure 1: Diagram of how the modules will be linked in Makefile

1.3 Module Interfaces

- **API of major module functions**
    - **Board.c**
        - Initial board
    - **Move.c**
        - Identifies a piece at a given position
        - Identify a piece color and return the character(w, b)
        - Undo move
        - Capture function (Optional, determined later)
        - Special cases:
            - Castling
            - En passant
            - Promotion
        - Check if move is legal and perform move
            - Pawn move
            - King move
            - Queen move
            - Rook move
            - Bishop move
            - Knight move
        - Win condition
            - Check
            - Stalemate
            - Checkmate
    - **AI.c**
        - AI
    - **MoveLog.c**
        - Writes to log
        - Save log

        - View log
    - **MoveList.c**
        - Move list
            - Doubly linked list struct implementation to store a list of moves. This list will support take back a move and move log feature

1.4 Overall program control flow

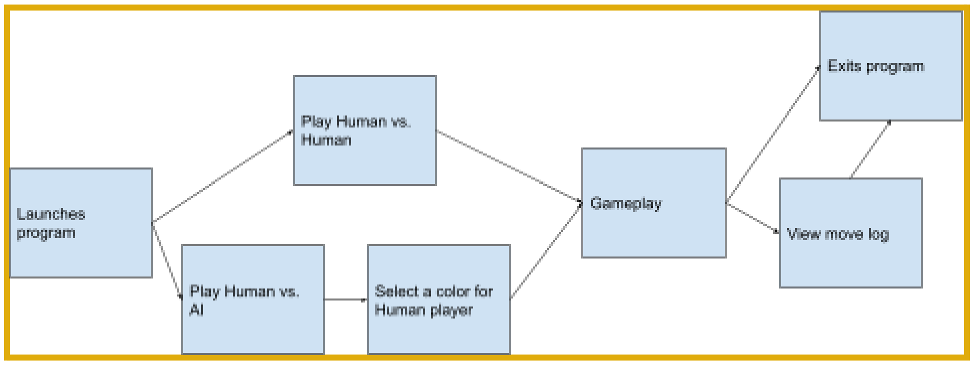
- 

Figure 2: User flow overview

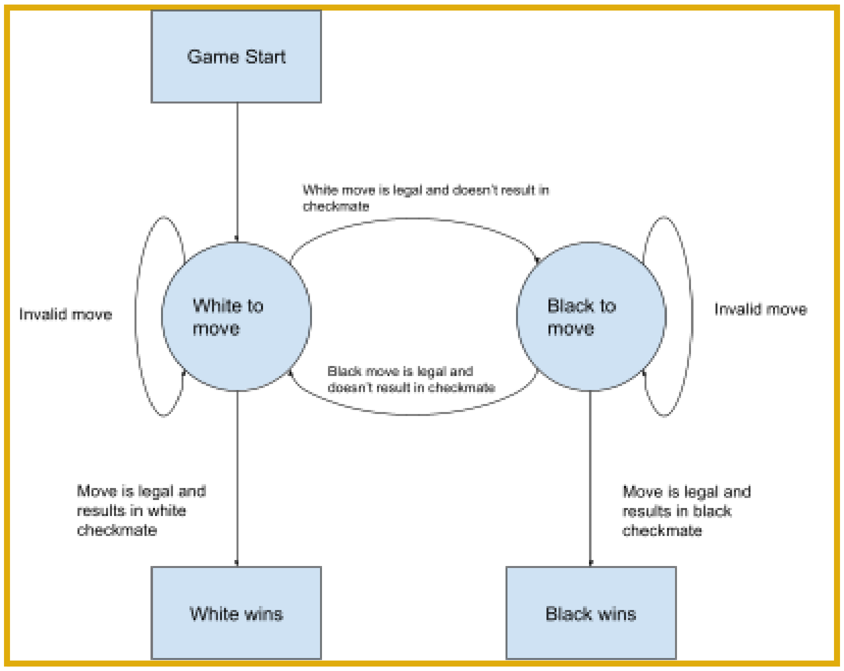
- 
Game Start
White move is legal and doesn't result in checkmate
Invalid move
White to move
Black to move
Invalid move
Black move is legal and doesn't result in checkmate
Move is legal and results in white checkmate
Move is legal and results in black checkmate
White wins
Black wins

Figure 3: Interaction flowchart once game has been started

-

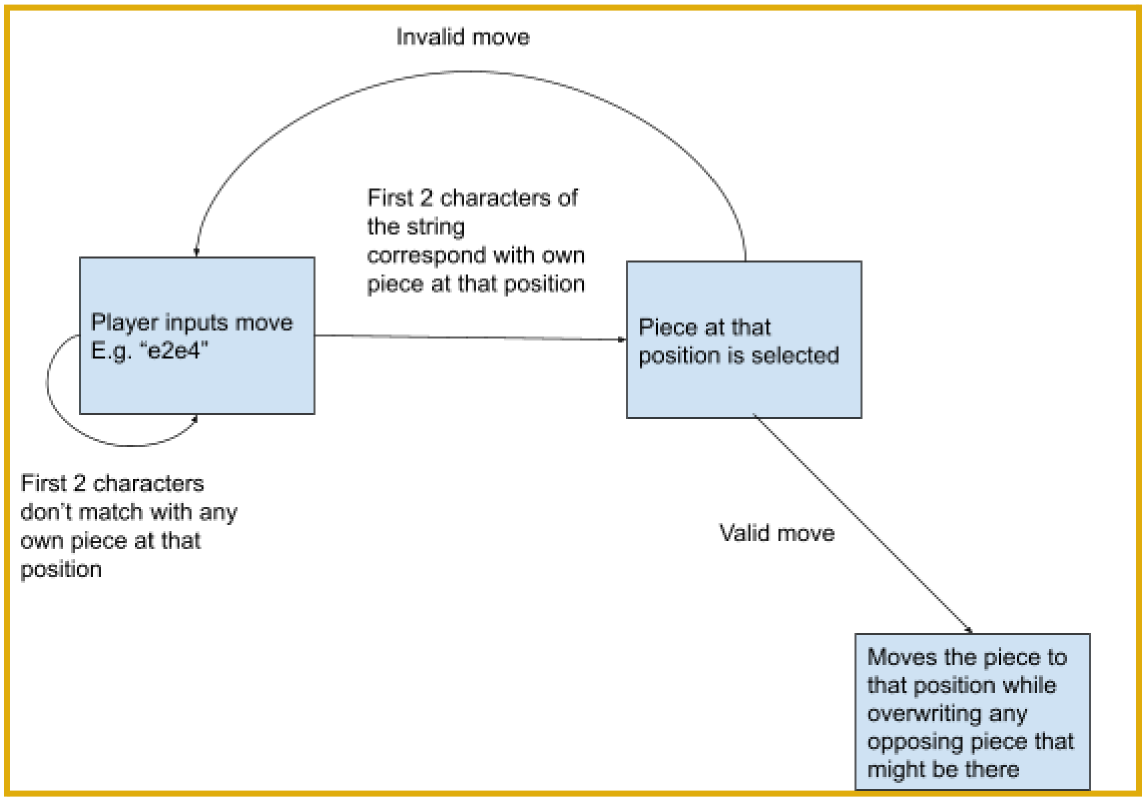

Invalid move
First 2 characters of the string correspond with own piece at that position
Player inputs move E.g. "e2e4"
Piece at that position is selected
First 2 characters don't match with any own piece at that position
Valid move
Moves the piece to that position while overwriting any opposing piece that might be there

Figure 4: Interaction flowchart when white or black inputs a move

2. Installation

2.1 System Requirements

- Recommended OS: Linux (CentOS release 6.10, Kernel: 2.6.32-754)
    - Contains a GCC/GNU compiler
- Memory: At least 512MB RAM
- Approximate Game Size: 1.03 Megabytes
- Dependencies: SDL (GUI to be implemented in later version), C standard libraries, Math library

2.2 Setup and Configuration

- **Install SDL**
  https://www.libsdl.org/download-2.0.php
- **Clone files from GitHub (will not need to be extracted)**
  git clone https://UCINETID@github.uci.edu/EECS-22L-S-21-Team-Projects/Team20.git

- **Get tar.gz archive**
  `cp ~Team20/bin/KGChess.tar.gz`

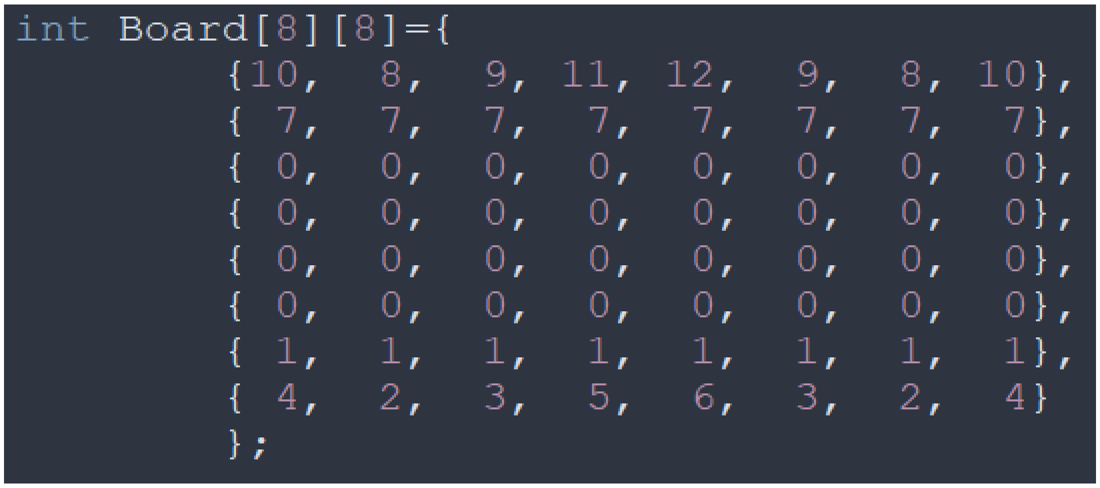
2.3 Building, compilation, installation

- **Extract tar.gz package**
  `gtar xvzf bin/KGChess.tar.gz`
- **Change current directory to KGChess**
  `cd KGChess`
- **Compile and generate executable**
  `make all`
- **Run program**
  `bin/KGChess`
- **Test piece rules (runs pieceTest.c)**
  `make test`
- **Test for memory leaks using Valgrind**
  `make memorytest`
- **Remove executable and object files**
  `make clean`
- **Remove all files (includes .c and .h files)**
  `make cleanall`

3. Documentation of packages, modules, interfaces

3.1 Detailed description of data structures

- **Board Structure**



```
int Board[8][8]={
            {10,  8,  9, 11, 12,  9,  8, 10},
            { 7,  7,  7,  7,  7,  7,  7,  7},
            { 0,  0,  0,  0,  0,  0,  0,  0},
            { 0,  0,  0,  0,  0,  0,  0,  0},
            { 0,  0,  0,  0,  0,  0,  0,  0},
            { 0,  0,  0,  0,  0,  0,  0,  0},
            { 1,  1,  1,  1,  1,  1,  1,  1},
            { 4,  2,  3,  5,  6,  3,  2,  4}
            };
```

Figure 5: Board Structure using 64 int 2D Array

  - The use of a 64 int 2-D array allows us to navigate through the board easily within code and by looking at it
    - Assign each number a piece and check each piece using a function that returns the piece character (e.g 0 = “ ” == Empty Space, 1 = wP == White Pawn)
  - The User/AI Input for the displayed board is saved to access later for the replay function in the struct’s char array
  - “Next” pointer points to the next object in the list
  - “Prev” pointer points to the prev object in the list, which allows an easier implementation of the take back move in the future

- **Piece Identity**
  - Piece type (Q, B, N, K, P, R)
  - Piece color (w or b before piece type)

3.2 Detailed description of functions and parameters

- **Functions in Board.h + Board.c modules**
  - void print_fun(int Board[8][8],….);
    - This function prints the board

            - int Board[8][8]: board of size 8x8. It may be an original board without any move or modified board with a given move

- **Functions in Move.h + Move.c modules**
    - **int IdentityDetect**(int Board[8][8], char Move[], int specifier)
        - This function will detect the piece at position origin or the piece at position destination depending on the specifier: (1 - origin, 2 - destination)
        - int Board[8][8]: board of size 8x8
        - char Move[]: 1D char array specifies the move of the piece to detect for its identity
        - int specifier: Specifies identifying space at the origin or the destination

    - **void pieceColor(int Board[8][8], char **Color);**
        - This function reads the numerical identity of the piece and changes the Color character to either w or b

    - **void Takeback(int Board[8][8], MLIST *MoveList)**
        - This function will take back a move if a human player asks for it.
        - A piece will be moved back to its previous position. And the current position will be set with its previous piece.
        - int Board[8][8]: board of size 8x8
        - MLIST *l: pointer to a list that keeps track of the moves.

    - **void Promotion(int Board[8][8], int color, int Rank, int File);**

    - This function will promote a pawn to another type of piece as the pawn reaches the end rank of the opponent's side.
    - int Board[8][8]: board of size 8x8
    - int color: this distinguishes whether the pawn being promoted is a black or white pawn.
    - int Rank: this takes in the rank of the pawn that is being promoted
    - Int File: this takes in the file of the pawn that is being promoted

- **void PawnMove(int Board[8][8], char Move[], MLIST *moveList, int promotionAllowed);**
    - This function will check whether a move is legal or not. If the move is legal this function will move a Pawn to the position specified by Move[]. This function will also support en passant and promotion
    - int Board[8][8]: board of size 8x8
    - char Move[]: 1D char array specifies a destination for the Pawn
    - MLIST *l: pointer to a list that keeps track of the moves.
    - int promotionAllowed: this takes in the account of whether promotion is allowed or not for the pawn. If allowed, then we use the Promotion function.

- **void KingMove(int Board[8][8], char Move[], int hasKingmoved);**
    - This function will check whether a move is legal or not. If the move is legal this function will move a King to the position specified by Move[].

    - This function also supports castling
    - int Board[8][8]: board of size 8x8
    - char Move[]: 1D char array specifies a destination for the King
    - int hasKingmoved: parameter to determine whether a king has moved previously

- **void QueenMove(int Board[8][8], char Move[]);**
    - This function will check whether a move is legal or not. If the move is legal this function will move a Queen to the position specified by Move[]
    - int Board[8][8]: board of size 8x8
    - char Move[]: 1D char array specifies a destination for the Queen

- **void RookMove(int Board[8][8], char Move[]);**
    - This function will check whether a move is legal or not. If the move is legal this function will move a Rook to the position specified by Move[]
    - int Board[8][8]: board of size 8x8
    - char Move[]: 1D char array specifies a destination for the Rook

- **void BishopMove(int Board[8][8], char Move[]);**
    - This function will check whether a move is legal or not. If the move is legal this function will move a Bishop to the position specified by Move[]
    - int Board[8][8]: board of size 8x8
    - char Move[]: 1D char array specifies a destination for the Rook

- **void KnightMove(int Board[8][8], char Move[]);**

        - This function will check whether a move is legal or not. If the move is legal this function will move a Knight to the position specified by Move[]
        - int Board[8][8]: board of size 8x8
        - char Move[]: 1D char array specifies a destination for the Knight

    - **int pieceMove(int Board[8][8], char Move[5], int hasKingmoved, MLIST *MoveList, int promotionAllowed);**
        - This function will check what piece is being moved and will return a corresponding piece function's move output
        - int Board[8][8]: board of size 8x8
        - char Move[]: 1D char array specifies a destination for the Knight
        - int hasKingmoved: parameter to determine whether a king has moved previously (only used for KingMove)
        - MLIST *l: pointer to a list that keeps track of the moves.
        - int promotionAllowed: this takes in the account of whether promotion is allowed or not for the pawn. If allowed, then we use the Promotion function.

    - **int winCon(int Board[8][8], char Move[5], MLIST *MoveList, int count);**
        - This function will check the board's state to determine if there is currently a check, stalemate, or checkmate calling from canCaptureKing()
        - int Board[8][8]: board of size 8x8

        - char Move[5]: Takes in the input of the user's input of the piece's origin they want to move and its destination.
        - MLIST *l: pointer to a list that keeps track of the moves.
        - int count: this counts the number of possible moves a piece around the king or the king itself can make when it is in check.

- **int canCaptureKing(int Board[8][8]);**
    - This function will determine if a given piece is able to capture the opponent's king. Basically a check function.
    - int Board[8][8]: board of size 8x8.

- **int IndexCheck(int x, int y);**
    - This function will determine if given coordinates x and y are in bound(from 0 to 7).
    - int x: x coordinate.
    - int y: y coordinate.

- **int OppKing(int Board[8][8], int Piece, int x, int y);**
    - This function will determine if 'Piece - bK or wK' is trying to attack the piece at position with x and y coordinates.
    - int Board[8][8]: board of size 8x8.
    - int Piece: attacker(maybe bKing or wKing depending on the scenario).
    - int x: x coordinate.
    - int y: y coordinate.

- **int OppBishop(int Board[8][8], int Piece, int x, int y);**

        - This function will determine if ‘Piece - bB or wB’ is trying to attack the piece at position with x and y coordinates.
        - int Board[8][8]: board of size 8x8.
        - int Piece: attacker(maybe bBishop or wBishop depending on the scenario).
        - int x: x coordinate.
        - int y: y coordinate.

    - **int OppRook(int Board[8][8], int Piece, int x, int y);**
        - This function will determine if ‘Piece - bR or wR’ is trying to attack the piece at position with x and y coordinates.
        - int Board[8][8]: board of size 8x8.
        - int Piece: attacker(maybe bRook or wRook depending on the scenario).
        - int x: x coordinate.
        - int y: y coordinate.

    - **int OppQueen(int Board[8][8], int Piece, int x, int y);**
        - This function will determine if ‘Piece - bQ or wQ’ is trying to attack the piece at position with x and y coordinates.
        - int Board[8][8]: board of size 8x8.
        - int Piece: attacker(maybe bQueen or wQueen depending on the scenario).
        - int x: x coordinate.
        - int y: y coordinate.

    - **int OppKnight(int Board[8][8], int Piece, int x, int y);**

        - This function will determine if ‘Piece - bN or wN’ is trying to attack the piece at position with x and y coordinates.
        - int Board[8][8]: board of size 8x8.
        - int Piece: attacker(maybe bKnight or wKnight depending on the scenario).
        - int x: x coordinate.
        - int y: y coordinate.

    - **int OppPawn(int Board[8][8], int Piece, int x, int y);**
        - This function will determine if ‘Piece - bP or wP’ is trying to attack the piece at position with x and y coordinates.
        - int Board[8][8]: board of size 8x8.
        - int Piece: attacker(maybe bPawn or wPawn depending on the scenario).
        - int x: x coordinate.
        - int y: y coordinate.

- **Functions in AI.h + AI.c modules**
    - **void AI(int Board[8][8],….);**
        - This function implements a random AI - AI selects a random valid move based on the first valid move available.
        - int Board[8][8]: original board of size 8x8.
        - AI will be using a index to keep track of the valid possible move s by mapping each column A-H
        - AI will use permutations in order to execute valid moves on the board

- **Functions in MoveList.h + MoveList.c modules**

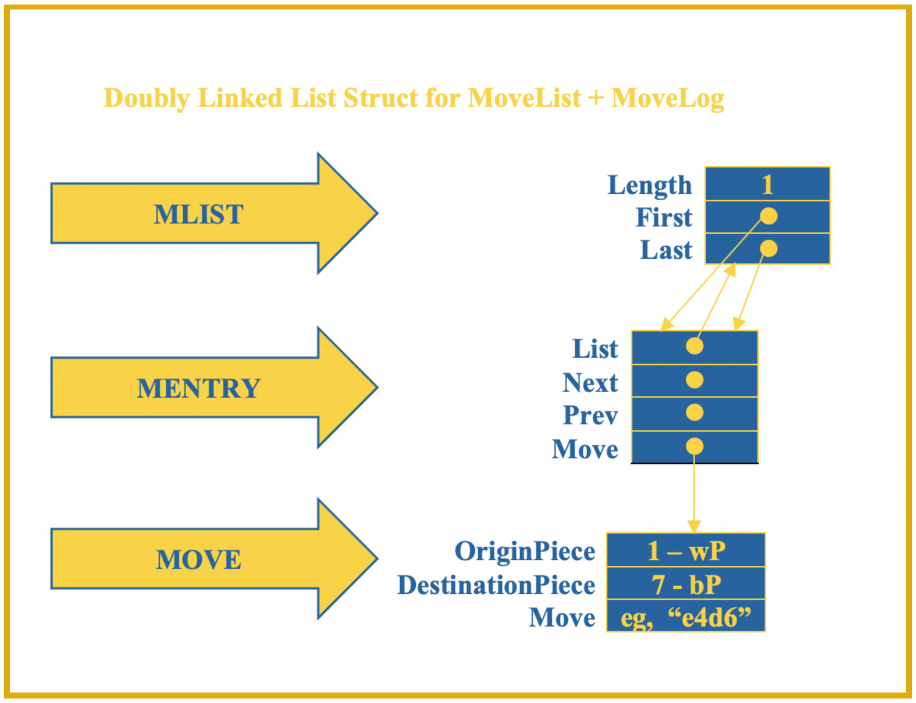


Figure 6: Diagram of Doubly Linked List (MoveList + MoveLog)

- **MOVE *NewMove(int OriginPiece, int DestinationPiece, char LTMove[5]]);**
  - This function allocates a new move record
  - int OriginPiece: takes in the piece at the origin.
  - int DestinationPiece: takes in the piece at the destination.
  - char LTMove[5]: an char array - User/AI input specifies a move. For example e4d6.

- **void DeleteMove(MOVE *m);**
  - This function deletes a move record
  - MOVE *m: a pointer to a node that holds a move. For example a pointer to a node that holds move e4d6.

- **MENTRY *NewMoveEntry(MOVE *m);**
  - This function allocates a new move entry

        - MOVE *m: a pointer to a node that holds a move. For example a pointer to a node that holds move e4d6.

    - **MOVE *DeleteMoveEntry(MENTRY *e);**
        - This function deletes a move entry
        - MENTRY *e: a pointer to a node that holds a pointer to the list, a pointer to the previous entry, a pointer to the next entry, a pointer to a node that holds a move.

    - **MLIST *NewMoveList(MLIST *l);**
        - This function allocates a move list
        - MLIST *l: a pointer to a list

    - **void DeleteMoveList(MLIST *l);**
        - This function deletes a move list
        - MLIST *l: a pointer to a list

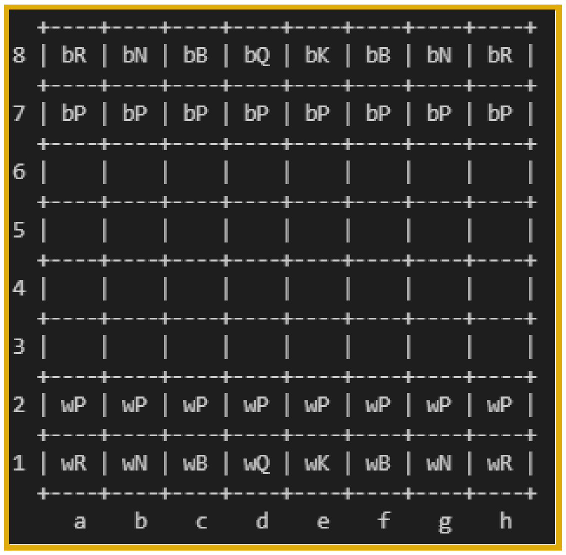
    - **void AppendMove(MLIST *l, MOVE *m);**
        - This functions appends a move at the end of list
        - MLIST *l: a pointer to a list
        - MOVE *m: a pointer to a node that holds a move. For example a pointer to a node that holds move e4d6.

    - **MOVE *RemoveLastMove(MLIST *l);**
        - This function removes the last move from the list
        - MLIST *l: a pointer to a list

- <u>**Functions in MoveLog.h + MoveLog.c**</u>
    - **void Print2File(char *fName, MLIST *l, int winner);**
        - This function prints a move log to a file

        - char *fName: File name
        - MLIST *l: Move list keeps track all the moves
        - int winner: 1 - white player is the winner, 2 - black player is the winner.
    - Void MoveLog (char *fName):
        - This function opens MoveLog from the saved file for the viewer to review.
        - fName: File's name that saves all the information about the previous game.

- <u>**Functions in KGChess.c**</u>
    - **void Human_Vs_Human(int Board[8][8],….);**
        - This function handles a Human v. Human game
        - int Board[8][8]: original board of size 8x8, with pieces at their original positions as shown below.
        - Input will be taken via char of size [5] with input examples such as "a2a3"
        - Board will be printed and updated from each players move
        - Parameters for legal moves will be handled in move.c
        - MLIST *Move_List: Is used when calling moves



```
  +----+----+----+----+----+----+----+----+
8 | bR | bN | bB | bQ | bK | bB | bN | bR |
  +----+----+----+----+----+----+----+----+
7 | bP | bP | bP | bP | bP | bP | bP | bP |
  +----+----+----+----+----+----+----+----+
6 |    |    |    |    |    |    |    |    |
  +----+----+----+----+----+----+----+----+
5 |    |    |    |    |    |    |    |    |
  +----+----+----+----+----+----+----+----+
4 |    |    |    |    |    |    |    |    |
  +----+----+----+----+----+----+----+----+
3 |    |    |    |    |    |    |    |    |
  +----+----+----+----+----+----+----+----+
2 | wP | wP | wP | wP | wP | wP | wP | wP |
  +----+----+----+----+----+----+----+----+
1 | wR | wN | wB | wQ | wK | wB | wN | wR |
  +----+----+----+----+----+----+----+----+
     a    b    c    d    e    f    g    h
```

Figure 7: Board printed at initial starting positions

- **int Exit_Undo(char uInpt[5], int count, int Board[8][8], MLIST *Move_List);**
    - This function allows the user to exit the game and undo a turn
    - char uInpt[5]: Checks if the input is “undo” or “exit”
    - int Board[8][8]: original board of size 8x8, with pieces at their original positions as shown below.
    - int count: The function uses it to check if the amount of turns is greater than 0 for undo
    - MLIST *Move_List: Is used when calling takeback function

- **void Human_Vs_AI(int Board[8][8],….);**
    - This function handles a Human v. AI game
    - int Board[8][8]: original board of size 8x8, with pieces at their original positions as shown previously.
    - Input will be taken via char of size [5] with input examples such as “a2a3”
    - Board will be printed and updated from each players move as well as Ai move
    - Ai will use the first valid legal move available
    - Parameters for legal moves will be handled in move.c
    - MLIST *Move_List: Is used when calling moves

- **void Ai_Vs_Ai(int board[8][8],.......);**
    - This functions handles a AI vs AI ga,e
    - int Board[8][8]: original board of size 8x8, with pieces at their original positions as previously shown
    - Input will be taken via char of size [5] with input examples such as “a2a3”
    - Board will be printed and updated from each AI’s move

        - Ai will use the first valid legal move available
        - MLIST *Move_List: Is used when calling moves
        - Parameters for legal moves will be handled in move.c

    - **void My_User_Input_Game(int count, char uInpt[5]);**
        - This function will prompt whoever's turn it is (White or Black) to move
        - int count: keeps track of whose turn it is
        - char Move[]: 1D char array with origin piece to destination location

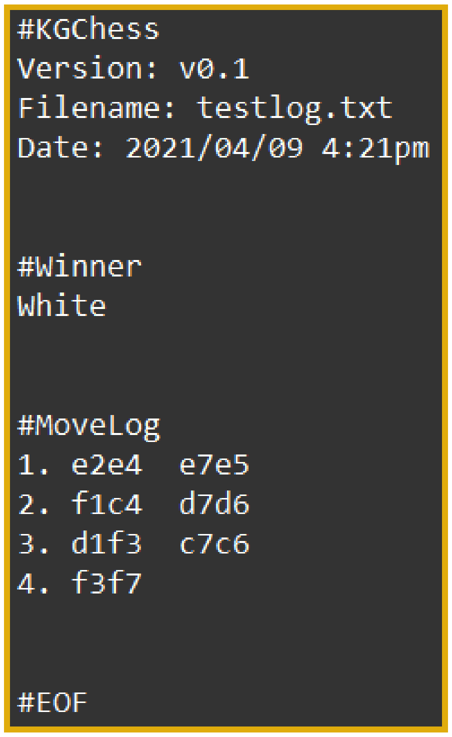
3.3 Detailed description of input and output formats

- **Syntax/format of a move input by the user**
    - If user wants to move piece from its original square to another square without violating any illegal moves, then user must first type in the rank and file of the original square, then the rank and file of the desired square:
        - Example: a pawn moving from e2 to e4
    - If the user wants to capture a piece, then follow the same format as moving a piece. This will only go through if the move is a valid move.
        - Example : a bishop from c1 capturing a black pawn on h6. Type in "c1 h6". If it is a legal move, then the program will capture. If not, an error message will occur.
    - If the user wants to do a special move like en passant, then just do the same thing as mentioned above. If it is a valid and legal move, then the move will be completed.
    - If the user wants to castle on the king's side, then the user needs to type in the king's spot on e1 and then the king's spot after the castle, which is g1.

        - Example: user wants to do king’s side castle: User inputs “e1 g1”. If it is possible and legal, then the move will perform. If not, an error message will occur.
            - If the user is black, then the same rules apply. Black user inputs “ e8 g8” to castle on the king’s side.
    - If the user wants to castle on the Queen’s side of the board, then the same rules apply as in the king’s side castle.
        - Example: user wants to do queen’s side castle: User needs to input “e1 c1”. If the move is possible and legal, then the move will perform. If not, an error message will occur.
            - If the user is black, then the same rules apply. Black user inputs “ e8 c8” to castle on the queen’s side.
- Syntax/format of a move recorded in the log file
    - The syntax for the moves recorded in the log file after the game will look like this.
        - Each turn will be recorded, as seen in the example, there will be numbers on the left hand side in order to count the number of moves each player took during the game.
        - There will be three columns:
            - The 1st column is the number of moves each player does.
            - The 2nd column is the move of white on that turn.
            - The 3rd column is the move of black on that turn.
        - The syntax for a move will be as follows:

- If a pawn moves, the log file will display the square the pawn moved to.
    - Ex: e2 to e4 is “e4”
- If the Bishop moves, the log file will display the abbreviation of the bishop and the square the bishop moved to.
    - Ex: Bishop moves from c1 to g5 is “Bg5”
- If the Knight moves, the log file will display the abbreviation of the knight and the square the knight moved to.
    - Ex: Knight moves from b1 to c3 is “Nc3”
- If the Rook moves, the log file will display the abbreviation of the rook and the square the rook moved to.
    - Ex: Rook moves from a1 to c1 is “Rc1”
- If the Queen moves, the log file will display the abbreviation of the queen and the square the queen moved to.
    - Ex: Queen moves from d1 to d2 is “Qd2”
- If the King moves, the log file will display the abbreviation of the king and the square the king moved to.
    - Ex: King moves from e1 to d2 is“Kd2”
- For a special move like a capture, there will be an x in between the abbreviation and the destination square.
    - Ex: Knight captures knight on e4 is “Nxe4”
- For a special move like a check, there will be a “+” at the end of the destination square.
    - Ex: Bishop checks King on b4 is “Be4+”

Here is an example of a log file in our program:



```
#KGChess
Version: v0.1
Filename: testlog.txt
Date: 2021/04/09 4:21pm


#Winner
White


#MoveLog
1. e2e4  e7e5
2. f1c4  d7d6
3. d1f3  c7c6
4. f3f7


#EOF
```

Figure 8: Screenshot of Log File

4. Development plan and timeline

4.1 Partitioning of tasks

- Tasks will be partitioned as evenly as possible so that each member will have a similar sized workload to prevent burnout, stress and promote a healthy work environment.
- Team Members will be able to volunteer for tasks and priority is given to members who request the task.
- In addition, experience and skill will be taken into account when assigning tasks if no member has volunteered for a given task. Example: Team Members with API experience will be asked to do the API task if no one elle volunteers since they would have the most experience in the given subject.

- Tasks will not be set in stone and Members may switch tasks or request assistance from other members and if outside issues such as sickness arise that will also be taken into account and tasks will be adjusted accordingly.

4.2 Team member responsibilities

- **Larrenz (Presenter):** King movement and canCaptureKing, debugging
- **Erick (Manager):** Pawn, Bishop functions, Main Menu, readability, and AI.c.
- **Tan (Recorder):** Work on main menu functionality, Queen, King, Pawn functions, special moves, AI, winCon, general debugging for all functions
- **Thanh (Reflector):** Board function, Bishop, Rook move function, canCapturedKing, IndexCheck, Opp(K, Q, B, Kn, P), Human vs Human, MoveList, MoveLog
- **Samuel (Reflector):** Knight function and Human vs Human, Debugging, Checking Valgrind, Optimizing functions, compacting functions

5. Back Matter

5.1 Copyright

5.2 References

Dang, Quoc-Viet. "EECS 22L, SPRING 2020, UNIVERSITY OF CALIFORNIA, IRVINE." UCI Canvas. canvas.eee.uci.edu/courses/36413.

"Design a Chess Game." *GeeksforGeeks*, 30 Sept. 2020, www.geeksforgeeks.org/design-a-chess-game/.

Yiran Zhong 2019 J. Phys.: Conf. Ser. 1195 012013

Index