CIS-17C

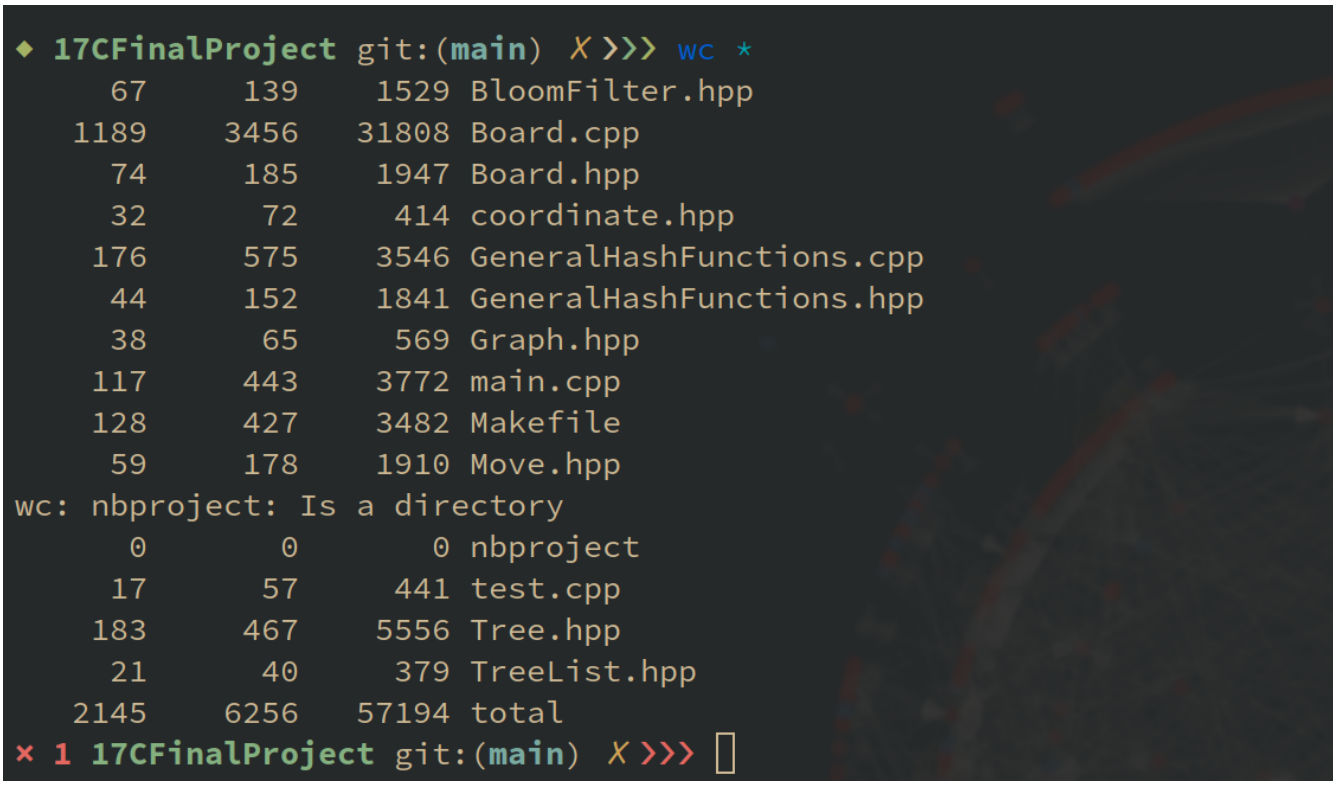
Final Project Write Up
- The Game of Chess

https://github.com/dabbingpenguin/17CFinalProject

Ansh Srivastava

Introduction

I had coded the chess game for my first project because I thought it would have been an interesting challenge because I would have to implement a lot more rules compared to some other games. I did not spend nearly as long on the second project as much as I did on the first project. The majority of my time was spent figuring out ways to implement the hashes, recursions, trees, and graphs, and debugging my code. It did not take that much time to code these elements in themselves. The project is located on https://github.com/dabbingpenguin/17CFinalProject . My project was around 2000 lines when I ran the wc command on it:



The code has 8 classes and 2 structs.

Approach To Development

My approach to development was to plan how to incorporate the recursions, graph, trees, and hashes before starting actually coding it. I broke the tasks down into small parts and kept a backup of each little part until I tested the code enough. For version control, I pushed my code to git every time I made major progress and kept copies of the code locally in case I ever wanted to revert back easily.

It did not take me a long time to implement the requirements. I thought that I could create a custom data type that could be indexed like an array or vector but is actually a tree. The way I could index a tree was to traverse through it and push the addresses of data for each node to a vector. I decided that I could use this custom datatype in a bloom filter that would allow me to fulfill the tree, and hashing requirement.

For the recursion requirement, I just converted my game while-loop to a recursive function. This was a little more difficult than I thought it would be because my loop had a lot of continue statements. Eventually I figured out that converting the continue statement to a recursive call followed by a return would do the trick. For the recursive sort, I decided to use merge sort on the list of captured

pieces. This would allow me to display captured pieces in based on how powerful they are if in the future I want to display captured pieces.

Implementing graphs was a little trickier. I figured that the only way I could efficiently use graphs in my chess game would be to represent moves by consider every square on the board a node and every legal move like a directional edge on a graph. My multiset of moves was mathematically the same structure as a graph because it could have been represented as a bunch of nodes connected with each other based on what moves are allowed. However, I decided to implement an adjacency list to get more practice with graphs and to get used to thinking of it in terms of nodes and edges.

Game Rules

The rules of the game are the typical rules of chess:

The game starts off with the position given on the left. Each coordinate on the board is given by a letter and a number for example the square on the bottom left of the board is a1 and the square on the top right of the board is h8. These coordinates are important because that is how the user interacts with the game in my project.

The pieces in my game are shown by two letter combinations. The first letter is for the player the piece is for, and the second letter is the type of piece it is.

Piece	Name
BP	Black Pawn
WP	White Pawn
WN	White Knight
BN	Black Knight
WB	White Bishop
BB	Black Bishop
WR	White Rook
BR	Black Rook
WQ	White Queen
BQ	Black Queen
WK	White King
WQ	White Queen

The pieces and their names are given in the table to the left. Here are the rules for how each type of piece can move:

- Rook: A rook can move either horizontally or vertically. It cannot jump over other pieces.
- Bishop: A bishop can move diagonally. Just like a rook, it cannot jump over other pieces.
- Knight: A knight can move 2 squares horizontally or vertically and 1 square perpendicular to it. A knight can jump over other pieces.
- Queen: A queen can move vertically, horizontally, and diagonally. It cannot jump over pieces just like the bishop and rook.
- King: A king can move just like a queen but one square at a time i.e., either 1 square horizontally, vertically, or diagonally.
- Pawn: A pawn can only move one square forward at a time. It cannot capture vertically tough. If there is an opponent's piece in front of your pawn, the pawn cannot capture the piece or move forward. The only way a pawn can capture is one forward diagonally or using en passant which will be explained later in the documentation. A pawn can also move 2 squares in its very first move.

Here are some additional rules that are just as simple:

- You cannot capture your own pieces.

- A check is defined as when the king is threatened and seems like it can be captured by the opponent.
- You cannot make a move that puts you in check.
- When you are in check, you must make a move that gets you out of check either by blocking , capturing the piece that is causing the check, or moving your king.
- If there are no ways to get out of check, it is defined as a checkmate and the player who checkmates their opponent wins the game.
- If you are not under check but do not have any legal move to make, it is considered a stalemate and the game is a draw.

Here are some additional special moves you can make in the game of chess:

- Castling: In order to castle, your king must not be under check.
  - Kingside Castling: Your king moves two squares towards the rook closest to your king and your rook moves one square to the left of the king. In order to do it, the squares your king will go through can not be threatened and you can not have moved your king or the rook you are castling with. In order to castle kingside, type in **0-0** when my game asks you for a move.
  - Queenside Castling: It is identical to kingside castling but the difference is that the king moves two squares to the left and the rook comes to one square right of the king. The same rules of not having moved your king or rook, and the squares the king moves through must not be threatened apply. In order to castle queenside, type **0-0-0** when my game asks you for a move.
- En passant pawn capture: This is a special pawn move. If your opponent just moved a pawn for the first time and decided to move 2 squares and you have a pawn right next to it, you can capture that pawn and move your pawn one square forward from where your opponent's square was. This move has to be done right after your opponent makes the pawn move. En passant can only be done immediately after the pawn move was made by the opponent. In order to en passant, type in **ENP** when my game asks for your move.

Other rules not implemented includes the ability to resign, offer draw, and the fifty-move rule.

Description of Code

Here is a description of all the source files in my code:

- coordinate.hpp - This header file has the coordinate structure defined. The coordinate structure just has an int for rank and one for file. It also has an anonymous enum that defines all of the files in the game - A,B,C,...,H that way I can treat files as integers too.
- Move.hpp - This header file has the Move struct defined. The move struct uses aggregation with two chess::coordinate structs. Most moves in my game are just treated as an initial square and a final square. If some piece is in the final square, it will be captured. The only functions in this struct are operator overloading for the comparison operators so that it works with sets.
- Board.hpp - This is the header for my chess::Board class. It deals with the storing the pieces in the board, captured pieces, storing whose move it is, storing which moves are legal, printing out the board to the screen, and checking when the game ends by checkmate or stalemate.
- Board.cpp - The definitions for the functions declared in my Board.hpp file belong here. This is the largest file in the project with 1089 lines and contains most of the logic.

- main.cpp - This file contains the code that executes when my game runs. It creates a chess::Board object in the main function and takes user input in a while loop (while the game is not over) and passes it into the chess::Board object.
- Graph.hpp - This file was used to create the adjacency list for my graph that would be used to represent legal moves.
- Node.hpp - This contained the Node struct that was used by the Tree class.
- Tree.hpp - This contained the templated tree class. This is the same as the class used for the everything trees assignment with the difference being an extra function to return the pointers to the data of each node to make it easier to index them.
- TreeList.hpp - This file contained my TreeList class - my custom data structure that would allow me to treat a tree like a list. It also has the [] operator overloaded.
- BloomFilter.hpp - This is almost the same as the class in my bloom filter assignment. A minor difference is that this class can now read/write to a file to save the bloom filter. The biggest difference is that I used my custom TreeList data structure in the bloom filter this time so I can utilize trees in my bloom filter.

The logic of most of the source files are self-explanatory. I shall describe how the code works using a flow chart and UML in a later section. Most of my class declarations were in header files with the .hpp file extension. When the class was not templated, I created a different .cpp source file for the classes.

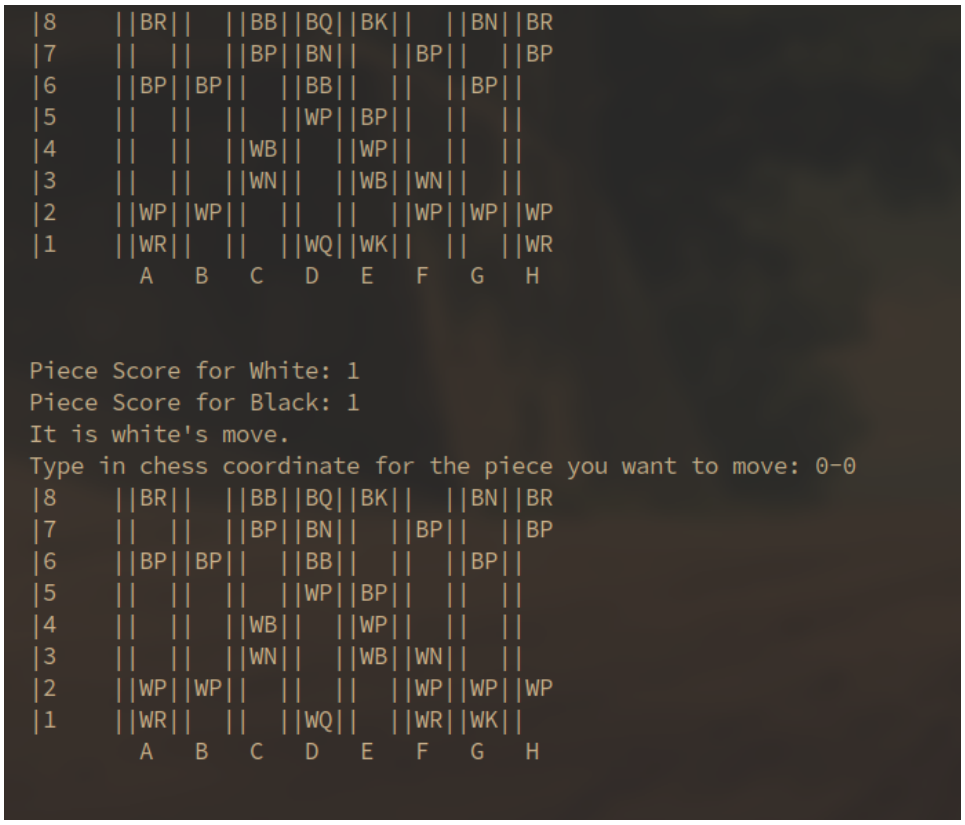
Sample Input/Output

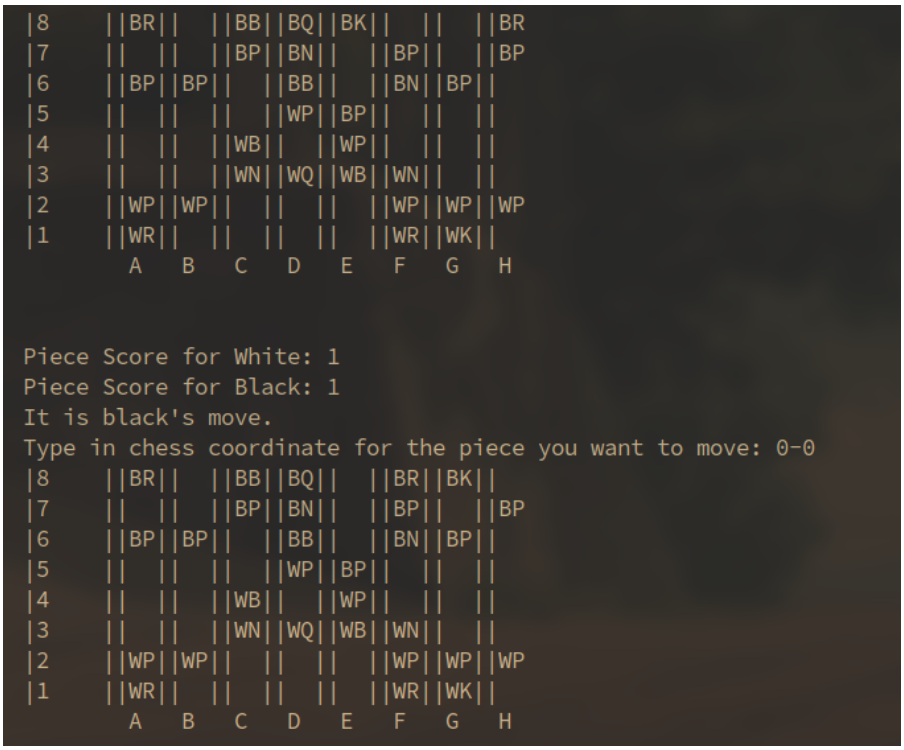
Here is a sample game of inputs and output:

```
|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


Piece Score for White: 0
Piece Score for Black: 0
It is white's move.
Type in chess coordinate for the piece you want to move: d2
Type in chess coordinate for the square you want to move your piece to: d4
|8     ||BR||BN||BB||BQ||BK||BB||BN||BR
|7     ||BP||BP||BP||BP||BP||BP||BP||BP
|6     ||  ||  ||  ||  ||  ||  ||  ||
|5     ||  ||  ||  ||  ||  ||  ||  ||
|4     ||  ||  ||  ||WP||  ||  ||  ||
|3     ||  ||  ||  ||  ||  ||  ||  ||
|2     ||WP||WP||WP||  ||WP||WP||WP||WP
|1     ||WR||WN||WB||WQ||WK||WB||WN||WR
         A   B   C   D   E   F   G   H


Piece Score for White: 0
Piece Score for Black: 0
It is black's move.
Type in chess coordinate for the piece you want to move: d7
Type in chess coordinate for the square you want to move your piece to: d5
|8     ||BR||BN||BB||BQ||BK||BB||BN||BR
|7     ||BP||BP||BP||  ||BP||BP||BP||BP
|6     ||  ||  ||  ||  ||  ||  ||  ||
|5     ||  ||  ||  ||BP||  ||  ||  ||
|4     ||  ||  ||  ||WP||  ||  ||  ||
|3     ||  ||  ||  ||  ||  ||  ||  ||
|2     ||WP||WP||WP||  ||WP||WP||WP||WP
|1     ||WR||WN||WB||WQ||WK||WB||WN||WR
         A   B   C   D   E   F   G   H
```

```
Piece Score for Black: 0
It is white's move.
Type in chess coordinate for the piece you want to move: c2
Type in chess coordinate for the square you want to move your piece to: c4
|8    ||BR||BN||BB||BQ||BK||BB||BN||BR
|7    ||BP||BP||BP||  ||BP||BP||BP||BP
|6    ||  ||  ||  ||  ||  ||  ||  ||
|5    ||  ||  ||  ||BP||  ||  ||  ||
|4    ||  ||  ||WP||WP||  ||  ||  ||
|3    ||  ||  ||  ||  ||  ||  ||  ||
|2    ||WP||WP||  ||  ||WP||WP||WP||WP
|1    ||WR||WN||WB||WQ||WK||WB||WN||WR
        A   B   C   D   E   F   G   H


Piece Score for White: 0
Piece Score for Black: 0
It is black's move.
Type in chess coordinate for the piece you want to move: d5
Type in chess coordinate for the square you want to move your piece to: c4
|8    ||BR||BN||BB||BQ||BK||BB||BN||BR
|7    ||BP||BP||BP||  ||BP||BP||BP||BP
|6    ||  ||  ||  ||  ||  ||  ||  ||
|5    ||  ||  ||  ||  ||  ||  ||  ||
|4    ||  ||  ||BP||WP||  ||  ||  ||
|3    ||  ||  ||  ||  ||  ||  ||  ||
|2    ||WP||WP||  ||  ||WP||WP||WP||WP
|1    ||WR||WN||WB||WQ||WK||WB||WN||WR
        A   B   C   D   E   F   G   H


Piece Score for White: 0
Piece Score for Black: 1
It is white's move.
Type in chess coordinate for the piece you want to move: e2
Type in chess coordinate for the square you want to move your piece to: e4
```

Here is a demonstration of kingside castling:





Here is what happens when you try to play an illegal move:

```
|8    ||BR||  ||BB||BQ||  ||BR||BK||
|7    ||  ||  ||BP||BN||  ||BP||  ||BP
|6    ||BP||BP||  ||BB||  ||BN||BP||
|5    ||  ||  ||  ||WP||BP||  ||  ||
|4    ||  ||  ||WB||  ||WP||  ||  ||
|3    ||  ||  ||WN||WQ||WB||WN||  ||
|2    ||WP||WP||  ||  ||  ||WP||WP||WP
|1    ||WR||  ||  ||  ||  ||WR||WK||
        A   B   C   D   E   F   G   H


Piece Score for White: 1
Piece Score for Black: 1
It is white's move.
Type in chess coordinate for the piece you want to move: c4
Type in chess coordinate for the square you want to move your piece to: b6
You made an illegal move. Try making your move again.
It is white's move.
Type in chess coordinate for the piece you want to move:
```

This is what a checkmate looks like:

```
|8    ||BR||BN||BB||BQ||BK||BB||BN||BR
|7    ||BP||BP||BP||BP||BP||  ||  ||BP
|6    ||  ||  ||  ||  ||  ||  ||  ||
|5    ||  ||  ||  ||  ||  ||BP||BP||
|4    ||  ||  ||  ||WP||WP||  ||  ||
|3    ||  ||  ||  ||  ||  ||  ||  ||
|2    ||WP||WP||WP||  ||  ||WP||WP||WP
|1    ||WR||WN||WB||WQ||WK||WB||WN||WR
        A   B   C   D   E   F   G   H


Piece Score for White: 0
Piece Score for Black: 0
It is white's move.
Type in chess coordinate for the piece you want to move: d1
Type in chess coordinate for the square you want to move your piece to: h5
|8    ||BR||BN||BB||BQ||BK||BB||BN||BR
|7    ||BP||BP||BP||BP||BP||  ||  ||BP
|6    ||  ||  ||  ||  ||  ||  ||  ||
|5    ||  ||  ||  ||  ||  ||BP||BP||WQ
|4    ||  ||  ||  ||WP||WP||  ||  ||
|3    ||  ||  ||  ||  ||  ||  ||  ||
|2    ||WP||WP||WP||  ||  ||WP||WP||WP
|1    ||WR||WN||WB||  ||WK||WB||WN||WR
        A   B   C   D   E   F   G   H


Piece Score for White: 0
Piece Score for Black: 0

Checkmate. White won!!!!!
◆ 17cprojectchess git:(master) ✗ >>>
```

Check Off Sheet

Sequences

A **std::list** is being used to store the captured pieces. The list is declared in the Board.hpp file:

```
std::list <int> capturedPieces;
```

Every time a piece is captured, it is added to the list in the Board.cpp file in the chess::Board::forceMove(Move mov) function:

```
std::swap(board[initialSquare.file][initialSquare.rank], board[finalSquare.file][finalSquare.rank]);
if (board[initialSquare.file][initialSquare.rank] != EMPTY)
  capturedPieces.push_back(board[initialSquare.file][initialSquare.rank]);
board[initialSquare.file][initialSquare.rank] = EMPTY;
```

Associative Containers

std::multiset

A **std::multiset<Move>** is declared in my Board.hpp file:

```
std::multiset<Move> legalMoves;
```

Whenever a move is legal, it is added to the legalMoves multiset in the chess::Board::fillset() function:

```
if (validateMove(mov))
{
  int a  = board[mov.firstSquare.file][mov.firstSquare.rank];
  int b = board[mov.lastSquare.file][mov.lastSquare.rank];
  if (currentMove.front() != pieceToPlayer(a))
    continue;
  board[mov.lastSquare.file][mov.lastSquare.rank] = a;
  board[mov.firstSquare.file][mov.firstSquare.rank] = chess::EMPTY;
  if ((currentMove.front() == Player::WHITE && !whiteIsUnderCheck()) || (currentMove.front() == Player::BLACK && !blackIsUnderCheck()))
    legalMoves.insert(mov);
  board[mov.firstSquare.file][mov.firstSquare.rank] = a;
  board[mov.lastSquare.file][mov.lastSquare.rank] = b;
}
```

std::map

Here is how I decided to use maps in my code:

```
std::pair<int, int> chess::Board::getPieceScore()
{
  std::map<int, int> pieceScores = {
      {WP, 1},
      {WR, 5},
      {WN, 3},
      {WB, 3},
      {WQ, 9},
      {BP, 1},
      {BR, 5},
      {BN, 3},
      {BB, 3},
      {BQ, 9}
    };
  int blackScore = 0;
  int whiteScore = 0;
  for (std::list<int>::iterator it = capturedPieces.begin();
      it != capturedPieces.end(); ++it)
  {
    if (pieceToPlayer(*it) == Player::WHITE)
      blackScore += pieceScores[*it];
    if (pieceToPlayer(*it) == Player::BLACK)
      whiteScore += pieceScores[*it];
  }
  std::pair <int, int> ret;
  ret.first = whiteScore;
  ret.second = blackScore;
  return ret;
}
```

```
int chess::getFileFromChar(char c)
{
  std::map<char, int> charToFile =
  {
    {'A', chess::A},
    {'B', chess::B},
    {'C', chess::C},
    {'D', chess::D},
    {'E', chess::E},
    {'F', chess::F},
    {'G', chess::G},
    {'H', chess::H},
    {'a', chess::A},
    {'b', chess::B},
    {'c', chess::C},
    {'d', chess::D},
    {'e', chess::E},
    {'f', chess::F},
    {'g', chess::G},
    {'h', chess::H}
  };
  return charToFile[c];
}

int chess::getRankFromChar(char c)
{
  std::map<char, int> charToRank =
  {
    {'1', 1},
    {'2', 2},
    {'3', 3},
    {'4', 4},
    {'5', 5},
    {'6', 6},
    {'7', 7},
    {'8', 8}
  };
  return charToRank[c];
}
```

```
chess::Player chess::pieceToPlayer(int piece)
{
  std::map<int, chess::Player> pToPlayer =
  {
    {chess::EMPTY, chess::Player::NOPLAYER},
    {chess::WP, chess::Player::WHITE},
    {chess::WB, chess::Player::WHITE},
    {chess::WR, chess::Player::WHITE},
    {chess::WN, chess::Player::WHITE},
    {chess::WK, chess::Player::WHITE},
    {chess::WQ, chess::Player::WHITE},
    {chess::BP, chess::Player::BLACK},
    {chess::BR, chess::Player::BLACK},
    {chess::BN, chess::Player::BLACK},
    {chess::BB, chess::Player::BLACK},
    {chess::BK, chess::Player::BLACK},
    {chess::BQ, chess::Player::BLACK}
  };
  return pToPlayer[piece];
}
```

Container Adapters:

std::stack

Board.hpp:

```
std::stack<Move> moves;
```

Board.cpp:

```
void chess::Board::forceMove(Move mov)
{
  // Check the rules
  // if (!validateMove(mov))
  //  return false;
  // Push to the stack
  moves.push(mov);
```

A std::stack was used to store the moves every player made. This is useful because it allows me to check if the previous move was a pawn move which is useful for en passant.

```
bool chess::Board::enpassant()
{
  if (!canEnpassant)
    return false;
  Move lastMove = moves.top();
```

and the lastMove variable is later used to perform all sorts of checks.

std::queue

Board.hpp:

```
std::queue<Player> currentMove;
```

and that is used in the constructor in Board.cpp:

```
currentMove.push(Player::BLACK);
currentMove.push(Player::WHITE);
```

whenever I need to check whose move it is, it uses *currentMove.front()*. In my makeAMove(Move mov) function, after every move, I change the move order like this:

```
currentMove.push(currentMove.front());
currentMove.pop();
```

Additional Containers:

std::pair

Pairs are used to store whether the players are allowed to castle in the chess::Board class:

```
std::pair <bool, bool> canCastleKingside;
std::pair <bool, bool> canCastleQueenside;
```

I found this more convenient than to have a different variables for each player.

2D std::vector

```
std::vector<std::vector<int>> board;
```

The 2D vector is declared in my chess::Board class. It is used to hold the piece in every square.

Iterators

Here are all the instances I used iterators in my Board.cpp file:

```
void chess::Board::printBoard()
{
    for (std::vector<std::vector<int>>::iterator it = board.end() - 1; it!= board.begin(); --it)
   {
      for (std::vector<int>::iterator jt = it->begin(); jt != it->end(); ++ jt)
      {
        int i = std::distance(board.begin(), it);
        int j = std::distance(it->begin(), jt);
        if (j==0)
        {
          if (i==8)
            std::cout << std::setw(2) << "|8" << std::setfill(' ');
          else
            std::cout << std::setw(2) << i << std::setfill(' ');
        }
        std::cout << std::setw(4) << getPieceName(board[j][i]) << std::setfill('|');
      }
      std::cout << '\n';
   }
  std::cout <<  "     " << std::setfill(' ')
```

```
bool chess::Board::blackIsUnderCheck()
{
  coordinate c;
  for (std::vector<std::vector<int>>::iterator it = board.begin() + 1;
      it != board.end(); ++it)
  {
    // for (std::vector<int>::iterator jt = it->begin() + 1;
    //      jt != it->end(); ++jt)
    std::vector<int>::iterator jt = std::find(it->begin(), it->end(), BK);
    if (*jt  == BK && jt != it->end())
      {
        c.file = it - board.begin();
        c.rank = jt - it->begin();
      }
  }
  return isSquareThreatenedForBlack(c);
}
```

```
bool chess::Board::whiteIsUnderCheck()
{
  coordinate c;
  for (std::vector<std::vector<int>>::iterator it = board.begin() + 1;
      it != board.end(); ++it)
  {
    // for (std::vector<int>::iterator jt = it->begin() + 1;
    //     jt != it->end(); ++jt)
    std::vector<int>::iterator jt = std::find(it->begin(), it->end(), WK);
    if (*jt  == WK && jt != it->end())
      {
        c.file = it - board.begin();
        c.rank = jt - it->begin();
      }
  }
  return isSquareThreatenedForWhite(c);
}
```

```
for (std::list<int>::iterator it = capturedPieces.begin();
    it != capturedPieces.end(); ++it)
{
  if (pieceToPlayer(*it) == Player::WHITE)
    blackScore += pieceScores[*it];
  if (pieceToPlayer(*it) == Player::BLACK)
    whiteScore += pieceScores[*it];
}
```

std::list, std::multiset, and std::map (eventough I did not need to use an iterator on map in my project) use bidirectional iterators. std::vector uses a random-access iterator. The difference between these iterators is that a bidirectional iterator can only increment or decrement by one element at a time but a random-access iterator can access any element in just a single step.
I did use std::find however in my code. It is supposed to return an input iterator. An input iterator is guaranteed to be able to increment but is not guaranteed to be able to decrement and may be read-only.

One thing to note is that a random access iterator is also a valid bidirectional iterator which is a valid forward iterator which is both a valid input and output iterator. All of these are valid trivial iterators because they may be de referenced. Here is a description of the types of iterators:

- Trivial Iterator- Is guaranteed to be able to be dereferenced.
- Input iterators - Is guaranteed to be able to incremented and read from.
- Output iterators - Is guaranteed to be able to incremented and written to.
- Forward iterator - Can be incremented, read, and written from.
- Bidirectional Iterators - Can be incremented, decremented, written, and read from.
- Random-Access Iterators - Can skip any number of elements to jump to another memory location in just one step.

std::list, std::multiset, std::map	**Bidirectional**, Forward, Output, Input, Trivial
std::vector	**Random-Access**, Bidirectional, Forward, Output, Input, Trivial

Algorithms

Non-mutating

std::distance

```
void chess::Board::printBoard()
{
    for (std::vector<std::vector<int>>::iterator it = board.end() - 1; it!= board.begin(); --it)
   {
      for (std::vector<int>::iterator jt = it->begin(); jt != it->end(); ++ jt)
      {
        int i = std::distance(board.begin(), it);
        int j = std::distance(it->begin(), jt);
        if (j==0)
```

^ std::distance was used because I needed the index of the element in my nested for loops.

std::find

```
bool chess::Board::blackIsUnderCheck()
{
  coordinate c;
  for (std::vector<std::vector<int>>::iterator it = board.begin() + 1;
     it != board.end(); ++it)
  {
    // for (std::vector<int>::iterator jt = it->begin() + 1;
    //     jt != it->end(); ++jt)
    std::vector<int>::iterator jt = std::find(it->begin(), it->end(), BK);
    if (*jt  == BK && jt != it->end())
      {
        c.file = it - board.begin();
        c.rank = jt - it->begin();
      }
  }
  return isSquareThreatenedForBlack(c);
}
```

```
bool chess::Board::whiteIsUnderCheck()
{
  coordinate c;
  for (std::vector<std::vector<int>>::iterator it = board.begin() + 1;
     it != board.end(); ++it)
  {
    // for (std::vector<int>::iterator jt = it->begin() + 1;
    //     jt != it->end(); ++jt)
    std::vector<int>::iterator jt = std::find(it->begin(), it->end(), WK);
    if (*jt  == WK && jt != it->end())
      {
        c.file = it - board.begin();
        c.rank = jt - it->begin();
      }
  }
  return isSquareThreatenedForWhite(c);
}
```

std::find is used to find the kings for purposes of checking for checks.

Mutating algorithms

std::swap

```
std::swap(board[initialSquare.file][initialSquare.rank], board[finalSquare.file][finalSquare.rank]);
if (board[initialSquare.file][initialSquare.rank] != EMPTY)
  capturedPieces.push_back(board[initialSquare.file][initialSquare.rank]);
board[initialSquare.file][initialSquare.rank] = EMPTY;
```

std::swap is used in my forceMove(Move) function to swap the values of two squared in the board.

Organization

std::sort

```
currentMove.push(Player::BLACK);
currentMove.push(Player::WHITE);
auto sortQueue = [](std::queue<Player> q)
{
    std::vector<Player> pVector;
    while (!q.empty())
    {
      pVector.push_back(q.front());
      q.pop();
    }
    std::sort(pVector.begin(), pVector.end(), [] (Player a, Player b)
        {
          if (a == Player::WHITE && b == Player::BLACK)
            return true;
          return false;
        });
    std::queue<Player> ret (std::deque<Player> (pVector.begin(), pVector.end()));
    return ret;
};
currentMove = sortQueue(currentMove);
```

In the constructor for my chess::Board class, I push the items in the wrong order for the currentMove queue. I then write a custom lambda function to sort it after converting it to a vector and then changing it back to a queue.

Recursion

The main loop for my game was implemented as a recursive function. In my midterm project, I had a while loop. I changed the while loop to a recursive function so I can practice recursion a little more. The function was void gameLoop(chess::Board&) in my main.cpp file. A screenshot of the function is on the next page.

```
void gameLoop(chess::Board &board)
{
  if (board.isGameOver())
    return;
  chess::coordinate initial;
  chess::coordinate finalcord;
  std::cout << "It is " << ((board.getPlayer() == chess::WHITE) ? "white" : "black" ) << "'s move.\n";
  std::cout << "Type in chess coordinate for the piece you want to move: ";
  std::string input;
  input = chess::cin();
  if (input == "0-0")
  {
    if (board.castleKingside())
    {
      board.printBoard();
      gameLoop(board);return;
    }
    else
    std::cout << "You cannot castle kingside right now.\n";
    gameLoop(board);return;
  }
  if (input == "0-0-0")
  {
    if (board.castleQueenside())
    {
      board.printBoard();
      gameLoop(board);return;
      }
    else
      std::cout << "You cannot castle queenside right now.\n";
      gameLoop(board);return;
    }
    if (input == "ENP")
    {
      if (board.enpassant())
      {
        board.printBoard();
        gameLoop(board);return;
      }
      else
        std::cout << "You cannot enpassant right now.\n";
      gameLoop(board);return;
    }
    if (input.size() != 2)
      std::cout << "We are unable to understand your input. Please enter a letter A-H followed by a number 1-8.\n";
    else{
      initial.rank = chess::getRankFromChar(input[1]);
      initial.file = chess::getFileFromChar(input[0]);
      if (initial.rank == -1 || initial.file == -1)
        {
          std::cout << "Not a valid coordinate!\n";
          gameLoop(board);return;
        }
      if (chess::pieceToPlayer(board.getPiece(initial.file, initial.rank)) != board.getPlayer())
      {
        std::cout << "It is " << ((board.getPlayer() == chess::Player::WHITE) ? "white" : "black") << "'s turn. Move your own piece.\n";
        gameLoop(board);return;
      }
      std::cout << "Type in chess coordinate for the square you want to move your piece to: ";
      std::string input;
      input = chess::cin();

      if (input.size() != 2)
        std::cout << "We are unable to understand your input. Please enter a letter A-H followed by a number 1-8.\n";
      else
      {
        finalcord.rank = chess::getRankFromChar(input[1]);
        finalcord.file = chess::getFileFromChar(input[0]);
        if (finalcord.rank == -1 || finalcord.file == -1)
          {
            std::cout << "Not a valid coordinate!\n";
            gameLoop(board);return;
          }
      }
    }
    chess::Move mov;
    mov.firstSquare = initial;
    mov.lastSquare = finalcord;
    if (!board.makeAMove(mov))
      {
      std::cout << "You made an illegal move. Try making your move again.\n"; gameLoop(board);return; }
    board.printBoard();
    gameLoop(board);
}
```

Recursive Sort

I have used merge sort to sort my list of captured pieces in my chess::Board class.

```
void chess::Board::sortCapturedPieces()
{
 const int sz = capturedPieces.size();
 int toSort[sz];
 std::copy(capturedPieces.begin(), capturedPieces.end(), toSort);
 mergeSort(toSort, 0, sz-1);
 while (!capturedPieces.empty())  capturedPieces.pop_front();
 capturedPieces.insert(capturedPieces.begin(), toSort, toSort + sz);
}
```

Basically, I used std::copy to convert the list into an array, sorted the array using mergesort, cleared the list, and then pushed all the elements back into the list once they were sorted.

Trees

I used the Tree class from the everything trees assignment, created a new modified function that is a modified version of my level-order traversal that would return pointers to the data stored in each node, and then created another data structure that was a wrapper for the tree but with the [] operator overloaded. I called that class TreeList. The class is very simple and I only added functions to push_back and the [] operator overload because that is all I needed for my bloom filter that I planned to use it for.

```
#ifndef TreeList_HPP
#define TreeList_HPP

#include <vector>
#include "Tree.hpp"

template <class T>
class TreeList
{
private:
  Tree<T> lstTree;
public:
  void push_back(T dat)   { lstTree.root = lstTree.insert(lstTree.root, dat); }
  T &operator[](int indx)
  {
    std::vector<bool*> indexes = lstTree.getIndexes(lstTree.root);
    return *(indexes[indx]);
  }
};

#endif
```

The code for Tree<T>::getIndexes is a modified version of level-order traversal that pushes the memory addresses to the vector.

```
std::vector<T*> getIndexes(Node<T> *tree){
    queue<Node<T>*> q;
    q.push(tree);
    std::vector<T*> res;
    while (!q.empty())
    {
        if (q.front()->left)      q.push(q.front()->left);
        if (q.front()->right)    q.push(q.front()->right);
         res.push_back(&(q.front()->data));
        q.pop();
    }
    return res;
}
```

Hashing

My BloomFilter class utilized my custom TreeList data type as well as hashing. The hash functions used were RSHash and FNVHash.

```
void addString(std::string str)
{
  bloomFilter[RSHash(str) % SIZE] = 1;
  bloomFilter[FNVHash(str) % SIZE] = 1;
  elementSize ++;
  writeToFile();
}

bool isProbableToExist(std::string str)
{
  return (bloomFilter[RSHash(str) % SIZE] && bloomFilter[FNVHash(str) % SIZE]);
}
```

I also added functionality to read and write to a file to save the bloom filter. My bloom filter was used in the main.cpp file to determine if someone had played the game before based on the name they provided.

```
  BloomFilter usernameBF(1024);
  std::cout << "Type in your username: ";
  std::string username = chess::cin();

  if (usernameBF.isProbableToExist(username))
  {
    std::cout << "You have probably played the game before based on our bloom filter (that
uses a tree).\n";
  }
  else
  {
    std::cout << "Our bloom filter indicate you have not played before.\n";
    usernameBF.addString(username);
  }
```

Graphs

I implemented a graph represented by an adjacency list in my Graph class.

Because the chess board has 64 squares, most legal moves can be represented by a starting square and a destination square. Another way to think of this is that each square is a node in a directional graph. I decided to use the Graph class to store the legal moves. I still kept the set of legal moves from the first project but I made sure to update my graph too. I used the graph to check if a move was legal instead of the multiset.

In chess::Board::makeAMove(Move), I used the isNeighbor function of the graph to determine if the move was legal.

```
#ifndef GRAPH_HPP
#define GRAPH_HPP

#include <list>

class Graph
{
private:
  int size;
  std::list<int> *adjacencylist;
public:
  Graph(int s)
  {
    size = s;
    adjacencylist = new std::list<int>[size];
  }
  void addDirectionalEdge(int a, int b)
  {
    adjacencylist[a].push_back(b);
  }
  void addEdge(int a, int b)
  {
    adjacencylist[a].push_back(b);
    adjacencylist[b].push_back(a);
  }

  bool isNeighbor(int node, int elem)
  {
    for (int x : adjacencylist[node])
    {
      if (x==elem)
        return true;
    }
    return false;
  }
};

#endif
```

```
int start = mov.firstSquare.getNum();
int dest = mov.lastSquare.getNum();
if (legalMovesGraph->isNeighbor(start, dest))
{
  forceMove(mov);
  moves.push(mov);
  currentMove.push(currentMove.front());
  currentMove.pop();
  fillSet();
  return true;
}
fillSet();
```

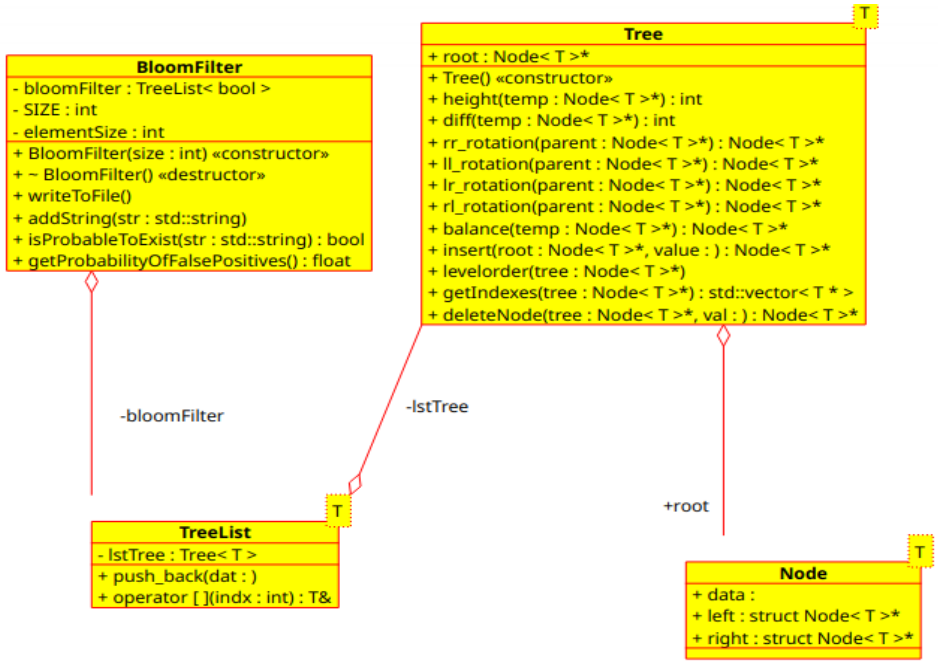
and to convert the multiset to the Graph, I did this in my chess::Board::fillSet function

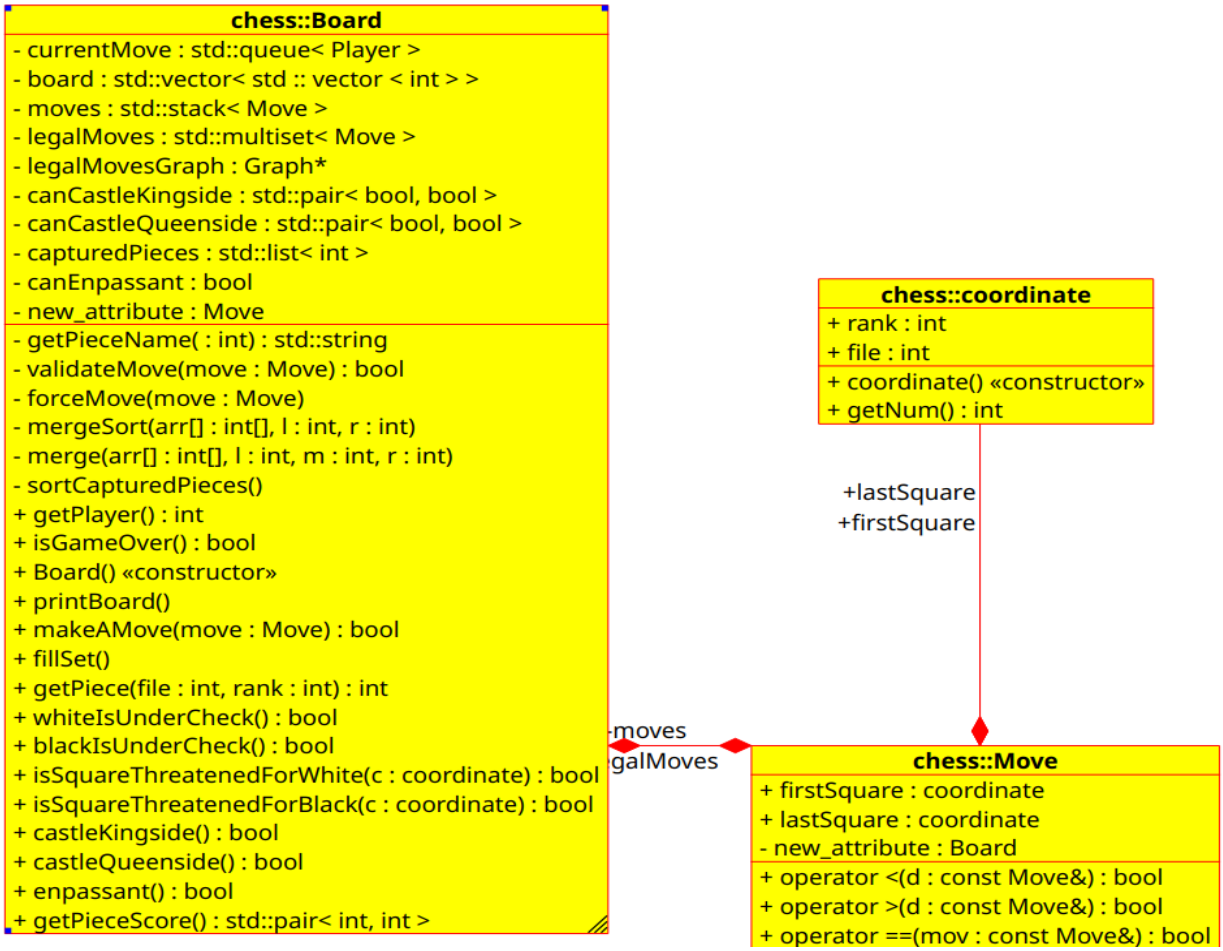
```
for (Move a : legalMoves)
{
  int start = a.firstSquare.getNum();
  int dest = a.lastSquare.getNum();
  legalMovesGraph->addDirectionalEdge(start, dest);
}
```

I also had to create a getNum() function in my coordinate struct so that I could convert every coordinate to a whole number so it could have been used as an index for my adjacency list.

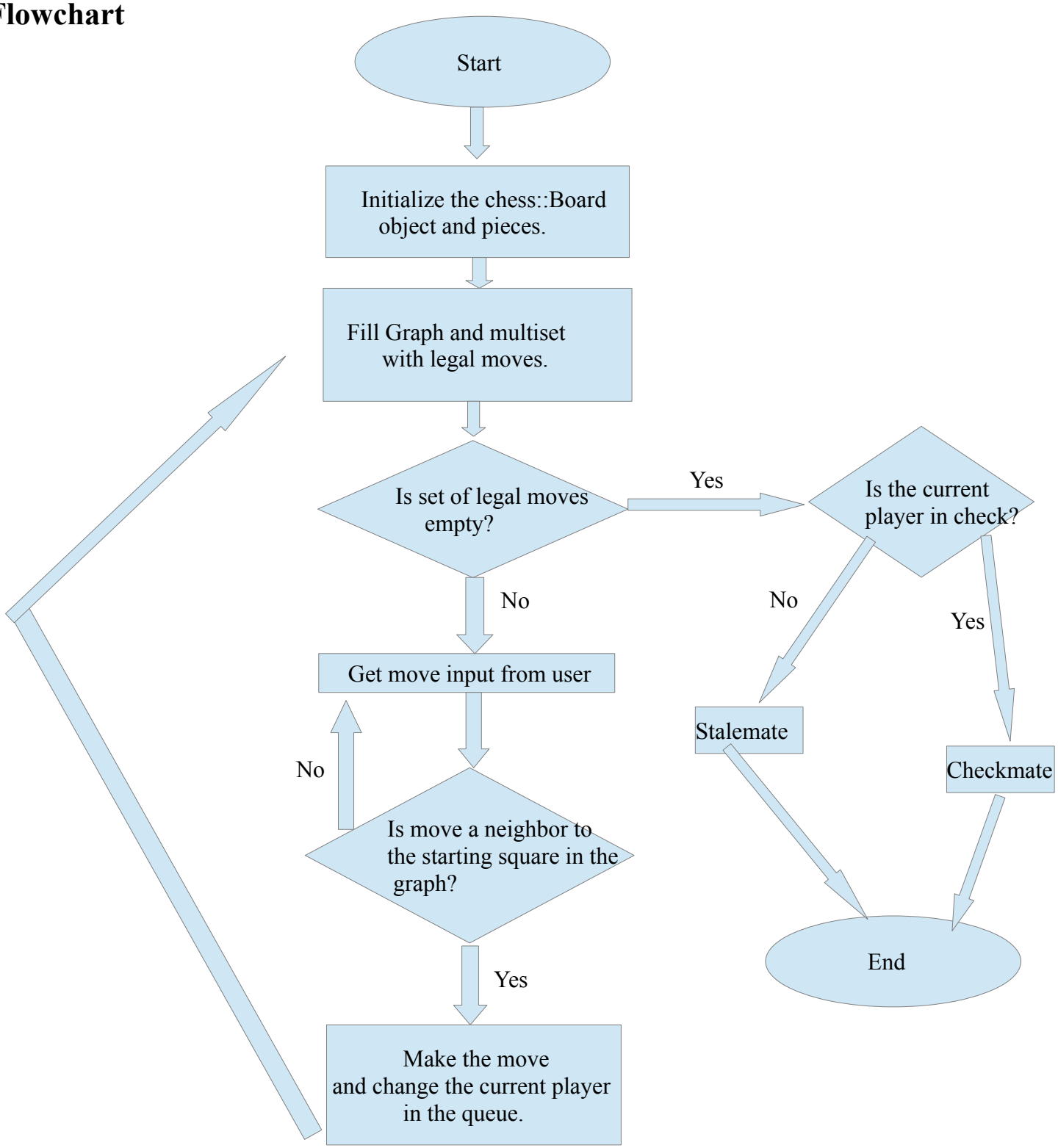
Documentation

UML


BloomFilter
- bloomFilter : TreeList< bool >
- SIZE : int
- elementSize : int
+ BloomFilter(size : int) «constructor»
+ ~ BloomFilter() «destructor»
+ writeToFile()
+ addString(str : std::string)
+ isProbableToExist(str : std::string) : bool
+ getProbabilityOfFalsePositives() : float
Tree
T
+ root : Node< T >*
+ Tree() «constructor»
+ height(temp : Node< T >*) : int
+ diff(temp : Node< T >*) : int
+ rr_rotation(parent : Node< T >*) : Node< T >*
+ ll_rotation(parent : Node< T >*) : Node< T >*
+ lr_rotation(parent : Node< T >*) : Node< T >*
+ rl_rotation(parent : Node< T >*) : Node< T >*
+ balance(temp : Node< T >*) : Node< T >*
+ insert(root : Node< T >*, value :) : Node< T >*
+ levelorder(tree : Node< T >*)
+ getIndexes(tree : Node< T >*) : std::vector< T * >
+ deleteNode(tree : Node< T >*, val :) : Node< T >*
-bloomFilter
-lstTree
+root
TreeList
T
- lstTree : Tree< T >
+ push_back(dat :)
+ operator [ ](indx : int) : T&
Node
T
+ data :
+ left : struct Node< T >*
+ right : struct Node< T >*


chess::Board
- currentMove : std::queue< Player >
- board : std::vector< std :: vector < int > >
- moves : std::stack< Move >
- legalMoves : std::multiset< Move >
- legalMovesGraph : Graph*
- canCastleKingside : std::pair< bool, bool >
- canCastleQueenside : std::pair< bool, bool >
- capturedPieces : std::list< int >
- canEnpassant : bool
- new_attribute : Move
- getPieceName(: int) : std::string
- validateMove(move : Move) : bool
- forceMove(move : Move)
- mergeSort(arr[] : int[], l : int, r : int)
- merge(arr[] : int[], l : int, m : int, r : int)
- sortCapturedPieces()
+ getPlayer() : int
+ isGameOver() : bool
+ Board() «constructor»
+ printBoard()
+ makeAMove(move : Move) : bool
+ fillSet()
+ getPiece(file : int, rank : int) : int
+ whiteIsUnderCheck() : bool
+ blackIsUnderCheck() : bool
+ isSquareThreatenedForWhite(c : coordinate) : bool
+ isSquareThreatenedForBlack(c : coordinate) : bool
+ castleKingside() : bool
+ castleQueenside() : bool
+ enpassant() : bool
+ getPieceScore() : std::pair< int, int >
chess::coordinate
+ rank : int
+ file : int
+ coordinate() «constructor»
+ getNum() : int
+lastSquare
+firstSquare
-moves
galMoves
chess::Move
+ firstSquare : coordinate
+ lastSquare : coordinate
- new_attribute : Board
+ operator <(d : const Move&) : bool
+ operator >(d : const Move&) : bool
+ operator ==(mov : const Move&) : bool

Flowchart


Start
Initialize the chess::Board object and pieces.
Fill Graph and multiset with legal moves.
Is set of legal moves empty?
Yes
Is the current player in check?
No
Yes
Stalemate
Checkmate
End
No
Get move input from user
No
Is move a neighbor to the starting square in the graph?
Yes
Make the move and change the current player in the queue.

Pseudocode

I am going to give psuedocode of how I fill up the set and graph with legal moves since that part is not clear from the flow chart.

CLEAR the multiset
CLEAR the graph
LOOP through i = 1 to 8 for all integer values of i
LOOP through j = 1 to 8 for all integer values of j
LOOP through k = 1 to 8 for all integer values of k
LOOP through l = 1 to 8 for all integer values of l
    Check if the piece in the square [i][j] can move to [k][l] according to the rules of movement of the piece
    if no, continue to the next iteration of the loop
    if yes, perform the move

    check if current player is under check
    if yes, add the move for [i][j]->[k][l] to the multiset of legal moves

    Reverse the move
LOOP through the set of legal moves
Add each combination of the starting and destination square to the LEGAL MOVES GRAPH as directional edges

The above is how I add legal moves to the set and then to the Graph. The graph is basically just a representation of the multiset (the multiset of moves is theoretically a graph by itself but it is not in the form of an adjacency list). I perform the move if it follows the rules of movement and then I check if the player would be in check after the move and then I reverse the move. If not under check, I add the move to the multiset. Then I reverse the move. When I fill up the multiset, I add the directional edges on the graph and then compare the graph against the move the user makes to determine if the move is legal.

It is still not clear how I check for checks or check if something follows the rules of movement. Here is an example for checking if a move follows the rules of movement for a bishop:

Given square [i][j] -> [k][l] we want to know if it is a valid move for a bishop
CREATE a variable dx = k-i
CREATE a variable dy = l-j
Is absolute value of dx = the absolute value of dy
If not then it is not a legal bishop move. RETURN FALSE
If yes, we still need to check if there is a piece in between [i][j]→[k][l]
CREATE a variable length=absolute value of dx or dy (because both will be the same value)
NORMALIZE dx, and dy by setting dx = dx/length and dy = dy/length
LOOP from n=1 to n=length-1
    Check if the square [i + n*dx][j+n*dy] is EMPTY
    If NOT EMPTY, RETURN FALSE
RETURN TRUE if it made it this far

And I use a similar process to check if the king is under check. I use the same method of using a normalized change in the x and y positions and loop through them to check if there is a bishop or a queen in each diagonal. I use a similar process for all other pieces.