Project 1

Chess

2003 Lines of Code

By James Rungsawang

Introduction

With most of the gameplay already finished, I tried to find ways to implement the new topics such as trees, graphs and recursion. This was mostly achieved providing information to the player such as who has captured what pieces or identifying the player's chained pawns.

Rules (Same as before)

The game contains the same rules as regular chess. All the pieces move the way they're supposed to and you are only allowed to make moves that obviously do not leave your king open to attack. The board has columns named by letters 'a' through 'h' and the rows labeled by numbers 1 - 8. There is no ai, so both players are controlled by typing in input.

Player 2 refers to white, pieces starting at the top of the board, and player 1 refers to black, pieces starting at the bottom of the board.

You first begin by entering the coordinates (I.E. a7) of the piece that you would like to move. Then you will be given all of the tiles that you will be able to move to from that piece. If you select the king, then you will also be given the option to castle from either side.

At any part of the game on any player's turn you may enter "captured" for a list of all the pieces that each player has captured (provided by two maps).

One of my favorite features that I've implemented was an undo system. If you type undo, it will revert the last move done by the last player and allow them to pick another move. This is done by having a stack containing a pair of two pairs. The pair first contains a pair of the coordinates of the piece before being moved and the data of the actual piece, the second pointer of the pair contains a pair of the coordinates of the piece after being moved and the data of the piece that occupied that space before the move was made.

The Game (Also same as before):

The board consists of an 8 by 8 2d array of data type Piece. Piece is a struct that contains:

String name: the type or name of the piece (I.E. queen, rook)

Int player: the player that it belongs to

Int moves: the number of moves this piece has made (needed for castling)

Bool en_passant: given to a pawn that has moved two spaces forward, is taken away if the opponent does not immediately capture

Check is calculated by determining if any piece is in range of the king and there are moves that the player can do to not lose the game. If the player has no available moves to remove the check (valid moves must not leave the king vulnerable) then checkmate is deemed and the game ends.

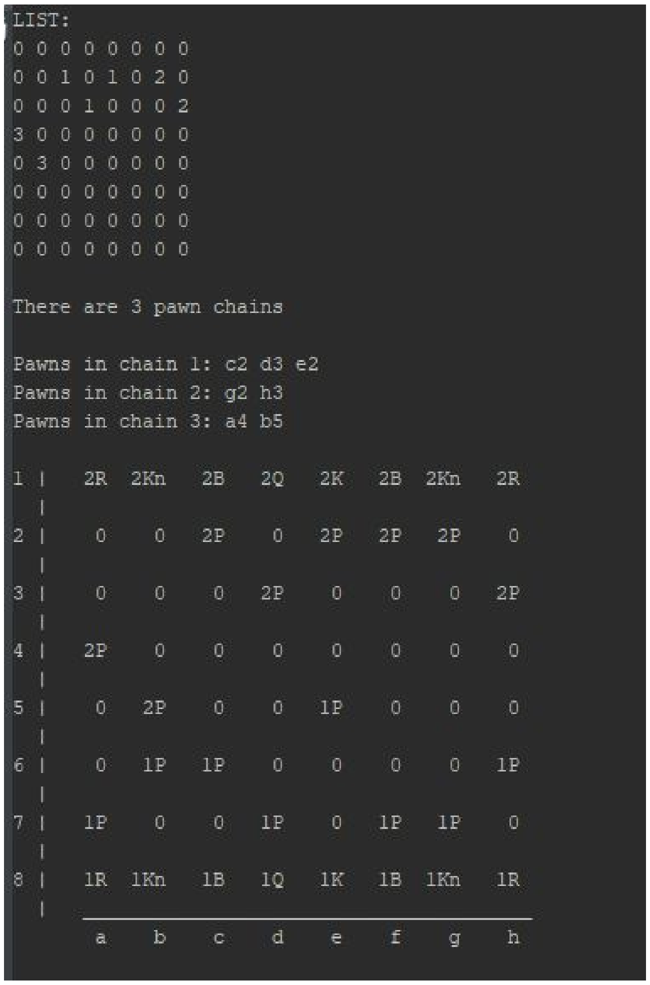
Implementation of New Topics

Graphs and Recursive Sort

In chess, a pawn chain is when multiple pawns are diagonal to each other. These are very strong as the pawns protect each other and are a key strategy to winning. I’ve decided to show the player how many pawn chains they have, how many pawns are in each chain, and which coordinates each of the pawns in the chain lie on. This was achieved by using a breadth first search.

I start by pushing a pawn into a queue. It then looks for any diagonal pawns and pushes those into the queue. If we find any chained pawns we push its x and y coordinates into a pair into a vector. All pawns belonging to the same chain will be given a unique number denoting their chain. Once we run out of neighboring pawns we have found the end of the chained pawns. This also marks the end of the breadth first search, when the queue runs out of pawn coordinates to check for. The total set of chained pawns is then pushed into a vector that will hold all of the chains.

Although all of the chains have been pushed into their respective chains, they are not sorted in order. I fixed this by calling a recursive bubble sort on the vector containing the chained pawns. Bubble sort iterates n-1 (the size of the array-1) times, each time swapping pairs of unsorted strings. Therefore we recursively call the bubble sort passing n as the number of iterations still needed to be done. The recursion ends when n is equal to 1.

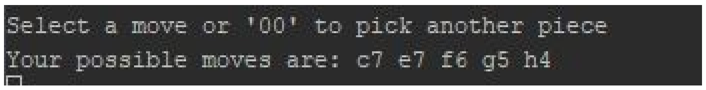


```
LIST:
0 0 0 0 0 0 0 0
0 0 1 0 1 0 2 0
0 0 0 1 0 0 0 2
3 0 0 0 0 0 0 0
0 3 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

There are 3 pawn chains

Pawns in chain 1: c2 d3 e2
Pawns in chain 2: g2 h3
Pawns in chain 3: a4 b5

1 |   2R  2Kn   2B   2Q   2K   2B  2Kn   2R
  |
2 |    0    0   2P    0   2P   2P   2P    0
  |
3 |    0    0    0   2P    0    0    0   2P
  |
4 |   2P    0    0    0    0    0    0    0
  |
5 |    0   2P    0    0   1P    0    0    0
  |
6 |    0   1P   1P    0    0    0    0   1P
  |
7 |   1P    0    0   1P    0   1P   1P    0
  |
8 |   1R  1Kn   1B   1Q   1K   1B  1Kn   1R
  |  ______________________________________
       a    b    c    d    e    f    g    h
```

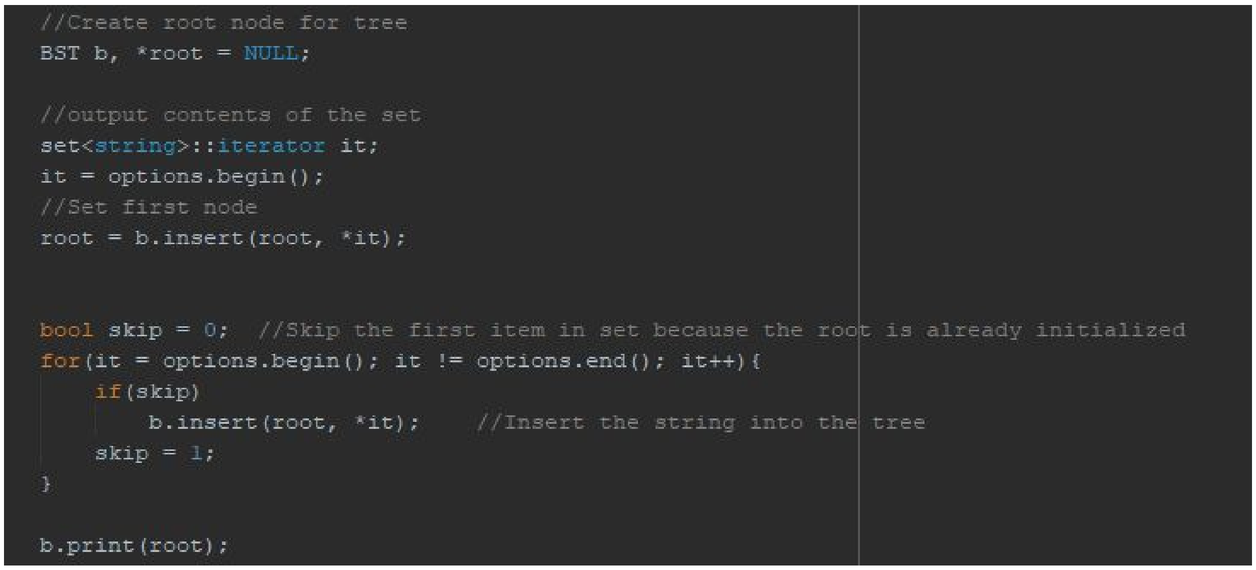
At the top of this image we can see that the pawns that are chained have the same numbers. Each cell that does not contain a chained pawn is marked as 0. Afterwards we write how many chains exist for this player and we output all of the pawn chains. Note how the pawns are listed in lexicographic order due to the recursive bubble sort.

Trees

So before, the player's options were stored in a set and then I would use an iterator to output the contents of the set. Instead, I have decided to store each of the player's options for a given piece as a binary search tree (BST). We begin by setting the root node as the first option and then we insert the other options. The tree will output the options in the lexicographically smallest order because it inserts each option to the left or right depending on if the string's value is smaller or greater. The resulting BST will contain all of the piece's options ordered alphabetically. Now I just output the tree in order to show the player what moves that piece can make.



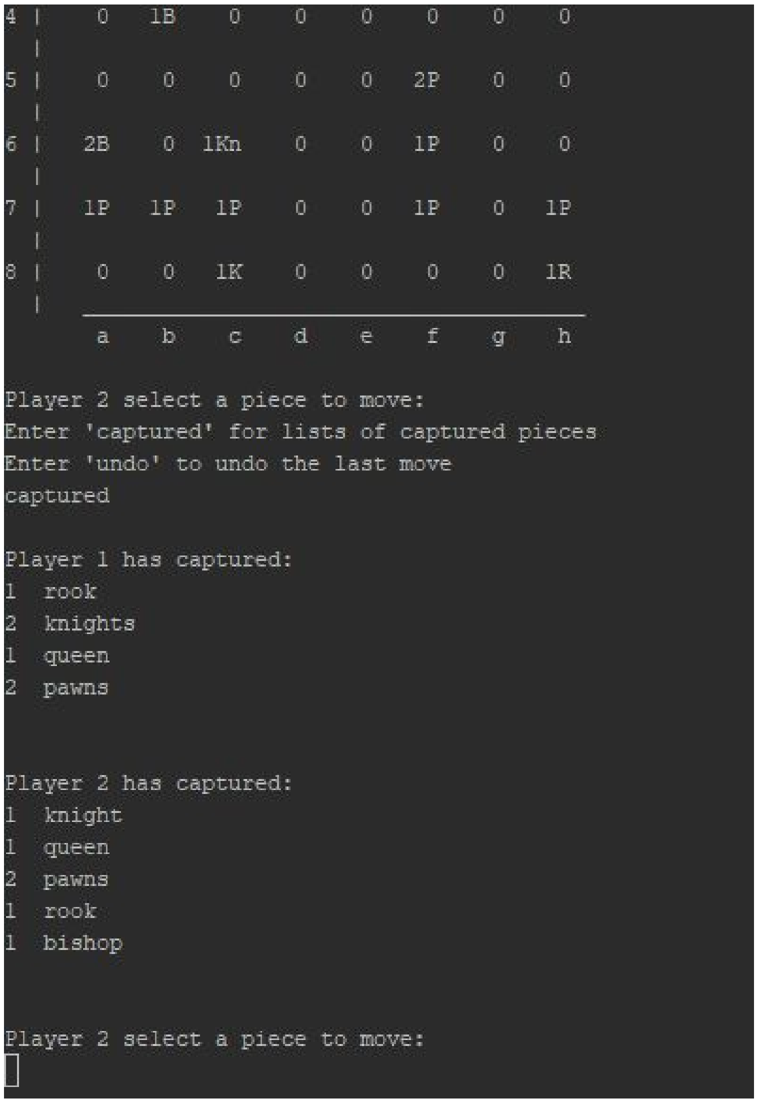
```
Select a move or '00' to pick another piece
Your possible moves are: c7 e7 f6 g5 h4
```



```
//Create root node for tree
BST b, *root = NULL;

//output contents of the set
set<string>::iterator it;
it = options.begin();
//Set first node
root = b.insert(root, *it);


bool skip = 0;  //Skip the first item in set because the root is already initialized
for(it = options.begin(); it != options.end(); it++){
    if(skip)
        b.insert(root, *it);    //Insert the string into the tree
    skip = 1;
}

b.print(root);
```

Hashing

The unordered_map in the STL is an implementation of a hash table. This is used to store which players have captured what pieces, and the frequency of said piece have been captured. The key for the hash table is simply the name of the piece and the value gets incremented each time that piece is captured. If the player types 'captured' on their turn, then this list of captured pieces will be shown.



```
4 |    0   1B    0    0    0    0    0    0
  |
5 |    0    0    0    0    0   2P    0    0
  |
6 |   2B    0  1Kn    0    0   1P    0    0
  |
7 |   1P   1P   1P    0    0   1P    0   1P
  |
8 |    0    0   1K    0    0    0    0   1R
  |     ______________________________________
       a    b    c    d    e    f    g    h

Player 2 select a piece to move:
Enter 'captured' for lists of captured pieces
Enter 'undo' to undo the last move
captured

Player 1 has captured:
1  rook
2  knights
1  queen
2  pawns


Player 2 has captured:
1  knight
1  queen
2  pawns
1  rook
1  bishop


Player 2 select a piece to move:
```