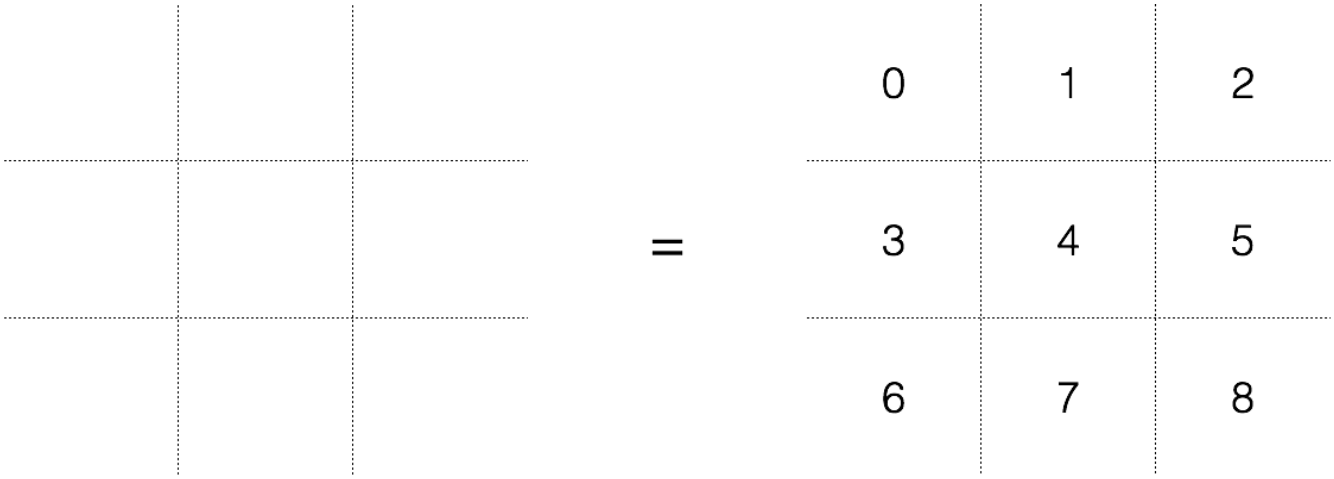
Project 5: Tic Tac Toe AI

In project one you will use the deque class you completed in homework five to create a class that plays Tic Tac Toe (X's and O's) optimally. The best strategy will be based on a game theory approach to the game. Make sure that you understand the rules and gameplay of the game before reading any further.

Part I: The Board

The first thing you need to play the game is a board. In our case you'll use the BoardXO struct. In C++ a struct can be used like a class where the data members are public by default. We'll do this because calculating the optimal play for a given game state will be memory/computationally intensive and we want to limit the overhead associated with manipulating the game state.

The Tic Tac Toe (T3) board consists of nine spaces (three rows and three columns). To minimize the time it takes to access the board and simplify your code, we'll refer to the board using linear indexing:

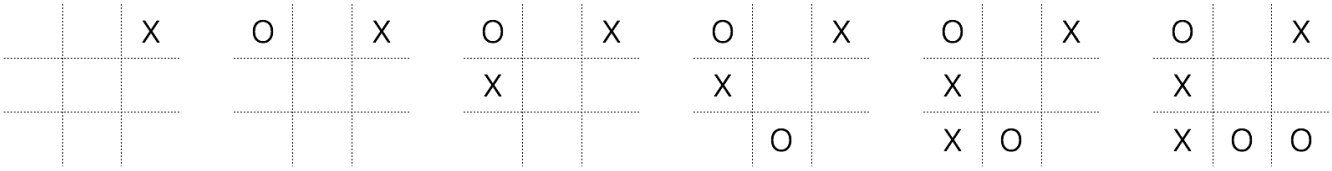


We'll denote the two players in the game, traditionally represented by X and O, as -1 and 1, respectively. That is X = -1 and O = 1. Through the course of the game these players place their markers (X and O) on the board. There are a maximum of nine total moves in the game and we'll refer to the moves using zero indexing (i.e., the first move will be the 0^{th} move the second move the 1^{st}, etc. This will make coding much easier).

BoardXO will consist of the following variables:

- `int8_t state[9]`: the current state of the game; i.e., where X and O have placed their markers. 0 denotes no marker has been placed in that position.
- `uint8_t moves[9]`: the position selected by the player for the 0^{th}, 1^{st}, 2^{nd}, 3^{rd}, etc. move
- `int8_t turn[9]`: the player responsible for the 0^{th}, 1^{st}, 2^{nd}, 3^{rd}, etc. move.
- `uint8_t numMoves`: the number of moves that have been made in the game

For example, the game:



would be result in the following:

- `int8_t state[9] = {1,0,-1,-1,0,0,-1,1,1}`
- `uint8_t moves[9] = {2,0,3,7,6,8,X,X,X}`
- `int8_t turn[9] = {-1,1,-1,1,-1,1,X,X,X}`
- `uint8_t numMoves = 6`

Note: here `X` represents an unknown/don't care value (i.e., you don't need to initialize these variables).

BoardXO will support the following methods:

- `BoardXO()`: the default constructor. The state variable should be initialized to all zeros (all positions unoccupied) and the number of moves (numMoves) to zero.
- `operator[]`: return the marker (player) occupying the position specified by the array access operator; e.g., assuming the variable `BoardXO brd` represents the above game, `brd[0] = 1`, `brd[1] = 0`, and `brd[2] = -1.`
- `operator<<`: output the current state of the game in X's and O's e.g., assuming the variable `BoardXO brd` represents the above game, `std::cout << brd << std::endl;` would produce:

```
O X
X
XOO
```

Note: BoardXO doesn't need to have methods to set/get variables as all of the variables are public (this is so we can access/set them without function call overhead)

Part II: The Game

Now that we have a way to denote the game state, it's time to define a class that allows us to manipulate the game state (i.e., make moves). The XO class keeps track of the state of the game using the private date member `BoardXO b`. We'll update the game state using the following methods:

- `XO()`: create a blank game (all positions unoccupied, no moves made).
- `XO(const int8_t* play, const uint8_t* pos, const uint8_t& n)`: initialize a game with a user specified state (board). The turns of the players is given in `play`, the positions they selected for their markers is given by `pos`, and the total number of moves/turns taken by `n`. (Note: these should be self-consistent and follow the rules of T3). For example, the above game could be initialized using `XO({-1,1,-1,1,-1,1}, {2,0,3,7,6,8}, 6)`. This method needs to update a board's state, moves, turn, and numMoves.
- `show()`: print the current state (board) to std::out
- `makeMove(const int8_t& play, const uint8_t& pos)`: place the marker for player specified by `play` at the board position `pos`. For example, assume that we want to make the first move of the above game. We would have: `XO g;` followed by `g.makeMove(-1,2);` This method needs to update a board's state, moves, turn, and numMoves.

- `makeMoves(const int8_t* play, const uint8_t* pos, const uint8_t& n)`: similar to the parameterized constructor above: make `n` moves given in the array `pos` for the players specified in `play`. This method needs to update a board's state, moves, turn, and numMoves.
- `makeOptimalMove()`: Determine the best play for whichever player plays next. Described below.
- `getBoard()`: return the current board. Useful for testing and grading.
- `gameOver()`: returns true if no more moves can be made; i.e., a draw has occurred or a player has won the game
- `winner()`: returns the player that has won the game (-1 or 1) or 0 if no player has won the game

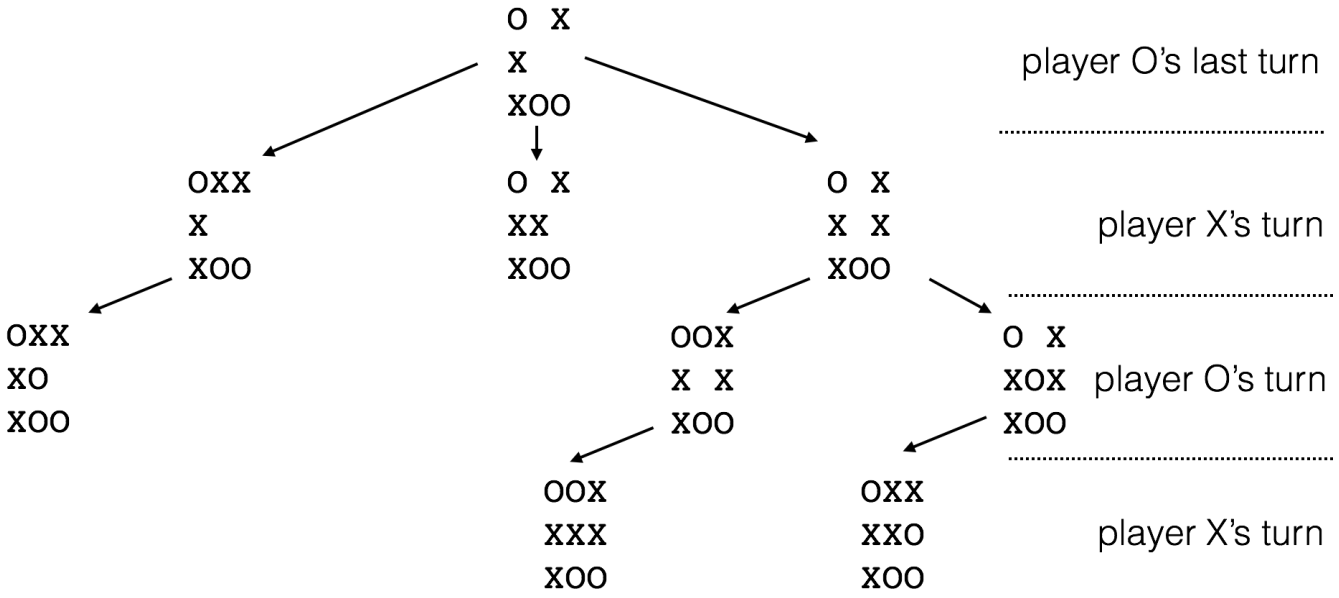
Part III: Finding Possible Moves

Completing this part will require that you understand and have implemented the above struct and class (except for `makeOptimalMove()`).

Let's consider the following use case: someone has built an interactive T3 game using our class. They're player O/1 and they've initialized the board to the game above. It's up to our class to find the best move for player X/-1 (the computer).

The first step in determining the optimal move is to enumerate all of the possible moves player X/-1 could make in response to player O's first move, then all of the possible moves that player O could make in response to player X's first move, etc. We'll accomplish this using breadth-first search (BFS). Many others use depth-first search (DFS) to solve this problem, even though BFS is faster, because BFS is memory intensive. If you're playing anything beyond checkers DFS is probably the right choice because BFS will exhaust the memory of your computer before an optimal solution is found. The state space of T3 is small so BFS is the way to go.

Let's use a tree to enumerate the possible moves that X/-1 could take and then that O/1 could take (this is called a move tree):



Your code will need to replicate the above and to do so you'll use a form of backtracking and a double-ended queue (deque). More specifically, you have to recreate the above tree using via BFS (this is a requirement of the project). In pseudo code this can be accomplished via:

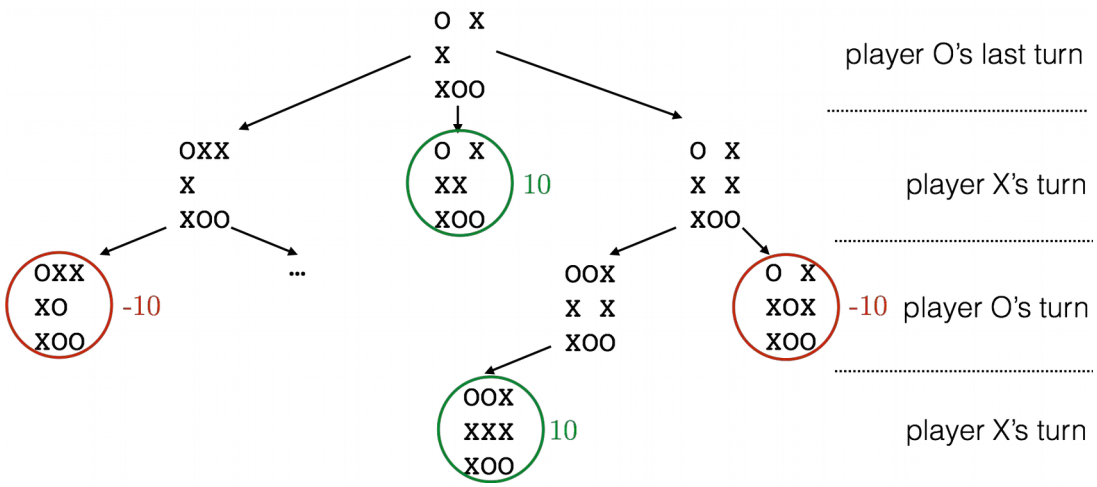
```
q is double-ended queue

push the current state (board) of the game to q

while q is not empty
  c = pop the game state from the front of q

  if the game c is not over
    push every possible move (board) for the next player to back of q
  else
    calculate the score of the initial move that led to win/lose/draw
```

Before you continue to the next part, you should implement the above in your makeOptimalMove() method, except for calculating the score.

Part IV: Scoring Moves

Let's continue with the game scenario above: one human player, O, versus the computer, X, and it's the computer's turn. To evaluate the strength of X's move we'll use what's known as minimax; i.e., we'll pick the move that minimizes the maximum score of the opponent. The minimax strategy has been shown to the optimal strategy for zero-sum games (one winner and one loser), like T3. We're also going to assume that both players are playing this strategy, even if the human doesn't play perfectly (optimally).

To keep things simple, let's assume that player O receives a score of -10 if they win the game (their maximum score), player X receives a score of 10 if they win the game (their maximum score), and a draw results in a 0 score. Given the game state above, player X has three possible moves (to positions 1, 4, 5). If we look at the tree below, we can see that for one of those moves player X will win (to position 4) and in two they will lose (to position 1, 5).



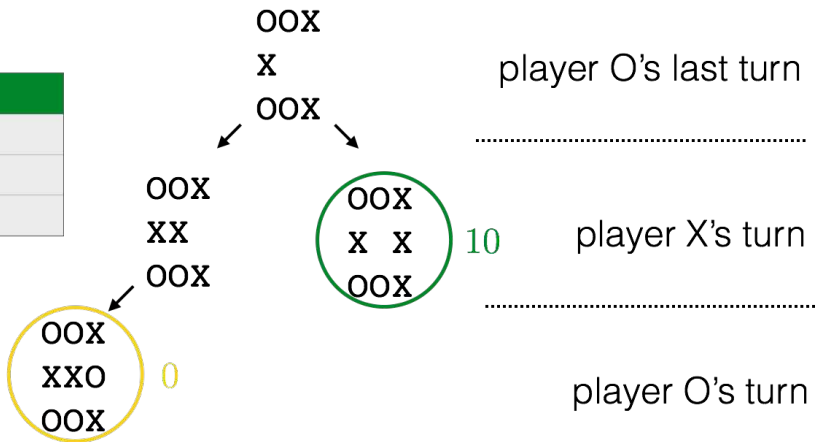
Note: some moves removed for the sake of clarity. If we were to tabulate the results:

Move	1	2	3
Position	1	**4**	5
Initial score	10	10	10
Final score	-10	10	-10

Remember, we're trying to minimize the opponent's, O's, maximum score (for the opponent -10 is their maximum because it means they won) so we would pick move 2.

An example move tree for a game that results in the possibility of a win or a draw for player X is:

Move	1	2
Position	4	**5**
Initial score	10	10
Final score	0	10



In this case player X would want to make the second move (to position 5). Notice that for both this case, and the one preceding it, we always take the maximum of the final scores (as this minimizes the opponents gain/score).

Since we're implementing the minimax strategy using BFS it's only necessary that we determine whether the opponent wins or draws (player O wins or draws). That is, we only need to answer the question "can the opponent win/draw if we make this move?" To answer this, I suggest that for the current player you create a list of scores for each possible initial move and assign a score of 10 to each (i.e., assume that each move will result in a win). An array or vector could be used to store the scores for each initial move. Upon reaching the game over state (a player won or a draw was reached) you have three conditions to consider:

1. if the current player (X in our example) won, do nothing
2. if the opponent (O in our example) won and -10 is less than the current score of the initial move then set the score for the initial move to -10
3. if the opponent (O in our example) caused a draw and 0 is less than the current score of the initial move then set the score for the initial move to 0

These three conditions should be considered for the line "`calculate the score of the initial move that led to win/lose/draw`" in the revised psueudo-code:

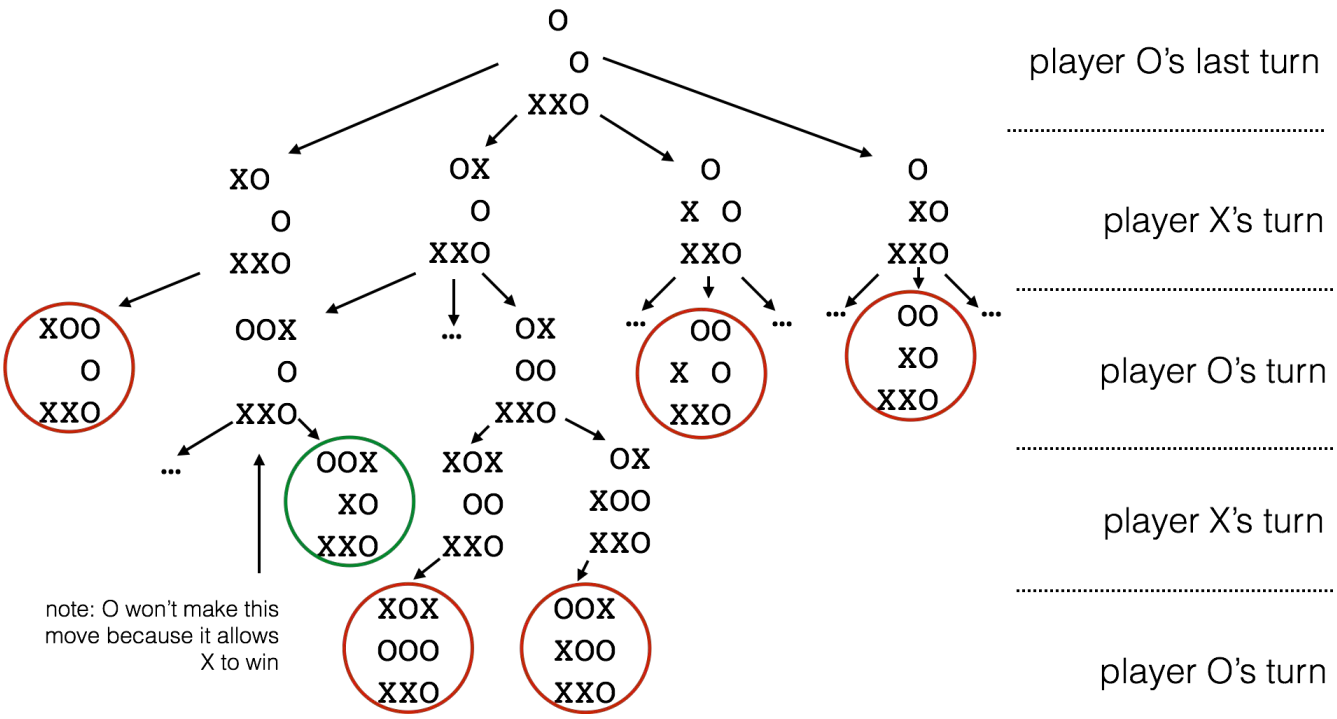
```
q is double-ended queue
s is a "list" of scores for initial moves by the current player

push the current state (board) of the game to q
```

```
while q is not empty
  c = pop the game state from the front of q

  if the game c is not over
    push every possible move (board) for the next player to back of q
  else
    calculate the score of the initial move that led to win/lose/draw

find the maximum score in s and make the corresponding move
```

Implement the above before moving onto the final part of the project.

Part V: Win fast, lose slow

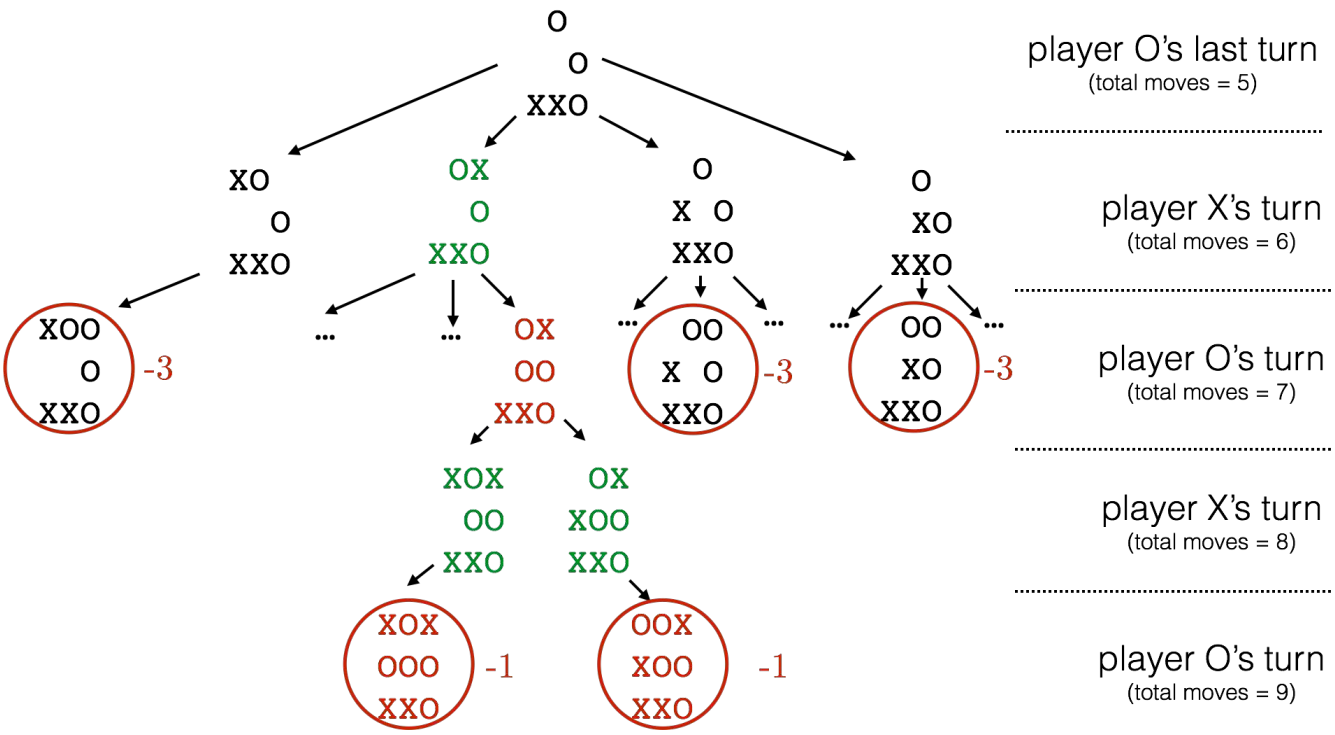
Consider the following scenario in which player X can only win if player O doesn't play optimally (as above it's player X's turn):



Now, the above scoring strategy (-10 for opponent win, 10 for player win) would result in player X making the move to position 0 and player O winning in the next round. Ideally, we'd like our opponent to have to play as many rounds as possible in the hope that they'd make a mistake. Conversely, if we can win a game we'd like to do it in as few as moves as possible. To accomplish this we'll need to modify our scoring strategy thusly:

if player wins: score = 10 – total number of moves
if opponent wins: score = total number of moves – 10

For example, in the above scenario the optimal move for player X to take is to position 2:



This is clear when we tabulate the scores because, remember, we're always picking the maximum of the final score so as to minimize the opponent's score (the opponent wants the most negative score possible):

Move	1	2	3	4
Position	0	**4**	3	4
Initial	1	1	1	1
Final	-3	-1	-3	-3

To implement the above scoring strategy we'll have to modify our three scoring conditions. First, the list of scores for each possible initial move should be assigned a default score of 1 (not 10 as in Part IV). Next, whenever the game over state is reached calculate the score according to whether the current player or their opponent won, factoring in the total number of moves, and then consider the following:

1. if the current player (X in our example) won and the current score of the initial move is > 0 (meaning the play could result in a win for the player) and the new score is greater than the current score for the initial move, then set the score for the initial move to the new score
2. if the opponent (O in our example) won and the new score is less than the current score of the initial move then set the score for the initial move to the new score
3. if the opponent (O in our example) caused a draw and 0 is less than the current score of the initial move then set the score for the initial move to 0