CS 32 – Project 3 Report: Michael Lui

1) A description of the design of your classes.

Board:

The Board class contains a single link list with multiple data types (deeper description of the data structure is given below) with multiple functions to operate and determine different states of the board such as move sow and set beans distinguishing number of beans and $_{number}$ of holes. The constructor takes the parameters of how to create a data structure and does so based on the parameters. Also two pointers of head and tail represent the linked list data structure of North and South side for the linked list.

Game:

Game class contains a private data member Board that is a shallow copy meaning that when operated on will not change the board that is passed into the constrictor. The class also has two private pointers for player that represent the player to get a move to play the game. The game class has the play function that simulates playing the game that checks certain standards for the rules of the mancala gave which are also checked in the move function which simulates picking a move from the player type.

Player:

Player class is a base class that is pure virtual meaning it cannot be created by itself. It has multiple functions to determine if it is a human or robot player.

Human Player:

Human Player is a derived class from Player overrides the move function by taking a human input.

Bad Player:

Human Player is a derived class from Player overrides the move function by taking sequentially the move that is valid.

Smart Player:

Human Player is a derived class from Player overrides the move function by taking smartest move (more detailed description given below).

We know what the public interfaces are, but what about your implementations:

User is welcome and prompted to enter variables for number of holes, beans per hole and type of player to start the game.

What are the major data structures that you use?

opted to use a single link list with multiple data types such as int id, int beans, Side side, Hole* across; Hole* next, and Hole* pot (names are explanatory for the purposes of the data types).

What private member functions or helper non-member functions did you define for what purpose?

Only private member function used was evaluated which was used for smart player move (implementation discussed below).

2) A description of your design for SmartPlayer::chooseMove, including what heuristics you used to evaluate board positions.

The chooseMove function is an algorithm based on depth breadth search on the tree of all possible game moves that is constrained with a max depth since getting to the bottom of the would be too computationally expensive.

chooseMove iterates through the for loop and if the move is valid calls upon the evaluate function to determine an ambiguous score IE a score weight based on the game board and the side.

Evaluate is a recursive algorithm that using depth first search algorithm to find and compare different score weights based on either the game ending (no more moves left) or the max depth has been reached.

This function iterates recursively by passing these data types

evaluate(Board b, Side s, int hole, int max, const Side actual)

Board b creates a new board (destructor will be called to prevent memory leaks)

Side s, determines the turn side as it keeps swapping each recursive iteration.

int hole, represents the best hole that is being iterated through (each recursive call has a for loop for every possible valid move that will return a score this will end when the base case of no more move or max depth is reach thus giving a weighted score)

The max game weight will be returned to the chooseMove function to determine the "evaluation" of a game move and thus in chooseMove the greatest weighted score move will be chose out of all possible moves.

3) pseudocode for non-trivial algorithms.

Board class:

//
Board(int nHoles, int nInitialBeansPerHole) // construction of board

```
m_holes = nHoles


head_S = new Hole
tail_N = head_S
```

```
head_S->beans = 0
head_S->id = 0
head_S->side = SOUTH;
head_S->pot = nullptr
head_S->across = nullptr

for (i = 0, i < nHoles;=, i++)
    Hole* temp = new Hole;
    temp->beans = nInitialBeansPerHole
    temp->side = NORTH
    temp->id = nHoles - i

    tail_N->next = temp
    tail_N = temp


// Create the North pot and South holes
head_N = new Hole
tail_S = head_N

head_N->beans = 0
head_N->id = 0
head_N->side = NORTH
head_N->pot = nullptr
head_N->across = nullptr

for (int i = 1; i <= nHoles; i++)
    Hole* temp = new Hole;
    temp->beans = nInitialBeansPerHole
    temp->side = SOUTH
    temp->id = i

    tail_S->next = temp
    tail_S = temp


// Link South pot to North last hole and North pot to South first hole
tail_S->next = head_S
tail_N->next = head_N

// Link across holes
Hole* temp_across1 = head_S->next
Hole* temp_across2 = head_N->next

for (i = 1, i <= nHoles, i++)
```

```
    for (int j = 1; j <= nHoles; j++) {=
        if (temp_across2->id == temp_across1->id)

            temp_across2->across = temp_across1;
            temp_across2->across->across = temp_across2;

        temp_across1 = temp_across1->next;



// Linking North holes to North pot
Hole* p = head_S->next
for (i = 0, i < m_holes, i++)
    p->pot = head_N;
    p = p->next;


// Linking South holes to South pot
p = head_N->next
for (int i = 0; i < m_holes; i++)
    p->pot = head_S
    p = p->next


//////////////////////////////////////////////
Board(const Board& RHS)

    //Initialize
    int nHoles = RHS's holes()
    int nInitialBeansPerHole = 0


    // Initialize number of holes
    m_holes = nHoles

    // Create the South Pot and the North holes
    head_S = new Hole
    tail_N = head_S

    head_S->beans = 0
    head_S->id = 0
    head_S->side = SOUTH
    head_S->pot = nullptr
    head_S->across = nullptr

    for (int i = 0; i < nHoles; i++)
```

```
    Hole* temp = new Hole
    temp->beans = nInitialBeansPerHole
    temp->side = NORTH
    temp->id = nHoles - i

    tail_N->next = temp
    tail_N = temp


// Create the North pot and South holes
head_N = new Hole
tail_S = head_N

head_N->beans = 0
head_N->id = 0
head_N->side = NORTH
head_N->pot = nullptr
head_N->across = nullptr

for (int i = 1; i <= nHoles; i++)
    Hole* temp = new Hole
    temp->beans = nInitialBeansPerHole
    temp->side = SOUTH
    temp->id = i

    tail_S->next = temp
    tail_S = temp


// Link South pot to North last hole and North pot to South first hole
tail_S->next = head_S
tail_N->next = head_N

// Link across holes
Hole* temp_across1 = head_S->next
Hole* temp_across2 = head_N->next

for (int i = 1; i <= nHoles; i++)
    for (int j = 1; j <= nHoles; j++)
        if (temp_across2->id == temp_across1->id)
            temp_across2->across = temp_across1
            temp_across2->across->across = temp_across2

        temp_across1 = temp_across1->next
```

```
    // Linking North holes to North pot
    Hole* p = head_S->next
    for (int i = 0; i < m_holes; i++)
        p->pot = head_N
        p = p->next


    // Linking South holes to South pot
    p = head_N->next
    for (int i = 0; i < m_holes; i++)
        p->pot = head_S
        p = p->next


    for (int i = 0; i <= m_holes; i++)

        setBeans(NORTH, i, RHS.beans(NORTH, i));
        setBeans(SOUTH, i, RHS.beans(SOUTH, i));
```

```
////////////////////////////////////////////
bool sow(Side s, int hole, Side& endSide, int& endHole)

    if (s == NORTH)
        Hole* temp = head_S->next;
        for (i = 0, i < holes, i++)
            if (temp->id == hole)
                break

            temp = temp->next
```

```
    if (temp->beans == 0 or temp->id == 0 or hole < 0 or hole > holes())
        return false


    int temp_beans = temp->beans
    temp->beans = 0
    temp = temp->next

    for (i = 0, i < temp_beans, i++)


        if (temp->id == 0 and s != temp->side)
            temp = temp->next
            i -= 1

        else

            temp->beans += 1;
            if (i != temp_beans - 1)

                temp = temp->next







    endSide = temp->side
    endHole = temp->id


    return true;
}

if (s == SOUTH)
    Hole* temp = head_N->next;
    for (int i = 0, i < holes, i++)


        if (temp->id == hole)
            break

        temp = temp->next
```

```
    if (temp->beans == 0 or temp->id == 0 or hole < 0 or hole > holes)
        return false


    int temp_beans = temp->beans
    temp->beans = 0
    temp = temp->next

    for (int i = 0, i < temp_beans, i++)


        if (temp->id == 0 and s != temp->side)
            temp = temp->next
            i -= 1

        else
            temp->beans += 1;
            if (i != temp_beans-1)

                temp = temp->next


    endSide = temp->side
    endHole = temp->id
    return true


  return false
```

Player class, Smart player:

```
//////////////////////////////////////////////
int chooseMove(const Board& b, Side s) const

  Initialize:
  int holes = b.holes()
  int bestHole = 1
  int bestScore = 1 // by default the 1st one should be chosen

  make Board copy a shallow copy of Board b

  // Iterate over each hole to find the best move
  for (hole = 1, hole <= holes, hole++)
```

```
    if (if b's beans at Side s and hole > 0)

        int score = evaluate(copy, s, hole, 0, s)

        // Update the best move if the current score is better
        if (score > bestScore)
            bestHole = hole
            bestScore = score



    return bestHole

//////////////////////////////////////////
int evaluate(Board b, Side s, int hole, int max, const Side actual) const


    if (b.beansInPlay(s) == 0 or max == 0) // ends game return pot num

        if (s == actual)

            Side endSide
            int endHole=-1
            for (int i = 1, i <= b.holes(), i++)

                Board temp(b)


                b.sow(actual, i, endSide, endHole);


            if (endHole == 0)

                return b.beans(s, 0)  + b.beansInPlay(actual) + 10



        Side opp2
        if (actual == NORTH)

            opp2=SOUTH;

        else

            opp2 = NORTH;
```

```
    return b.beans(s, 0)+b.beansInPlay(actual);



Side endSide
int endHole
b.sow(s, hole, endSide, endHole)

Side opp = actual
if (s == NORTH)

    opp = SOUTH

else

    s = NORTH;


// bean to empty spot case taking across hole to pot
if (b.beans(endSide, endHole) == 1 and b.beans(opp, endHole) != 0)

    if (s == NORTH)


        b.moveToPot(NORTH, endHole, NORTH)
        b.moveToPot(SOUTH, endHole, NORTH)

    else


        b.moveToPot(NORTH, endHole, SOUTH);
        b.moveToPot(SOUTH, endHole, SOUTH);



//capture case
if (b.beans(endSide, endHole) == 1 && b.beans(opp, endHole) != 0)

    if (s == NORTH)


        b.moveToPot(NORTH, endHole, NORTH);
        b.moveToPot(SOUTH, endHole, NORTH);
```

```
    else

        b.moveToPot(NORTH, endHole, SOUTH);
        b.moveToPot(SOUTH, endHole, SOUTH);



int bestScore = 1;
int score = 1;

for (int i = 1; i <= b.holes(); i++)

   if (b.beans(s, i) != 0)// possible move

      if (s == NORTH)

         score =evaluate(b, opp, i, max - 1, actual);


      else

         score =evaluate(b, opp, i, max - 1, actual);



      if (score > bestScore)


         bestScore = score;
```

Game class:

```
//////////////////////////////////////////
bool move(Side s) // check 0 bean case for a side

   bool over
   bool hasWinner
   Side winner
   status(over, hasWinner, winner)


   // Check if the game is already over
   if (over)
```

```
    return false;

int hole = -1

if (s == NORTH)


    hole = m_northPlayer->chooseMove(m_board, NORTH);
    print "hole chosen " + hole + end line

    if (m_northPlayer->isInteractive())

        print "(NORTH) turn" + end line
        print "Press ENTER to continue..." << end line
        Wait for ENTER key



else


    hole = m_southPlayer->chooseMove(m_board, SOUTH);
    print "hole chosen " + hole end line
    if (m_southPlayer->isInteractive())

        print "(SOUTH) turn" << end line
        print "Press ENTER to continue..." end line
        Wait for ENTER key


//move by sowing
Side endSide
int endHole
m_board.sow(s, hole, endSide, endHole)

Side opp
if (endSide == NORTH)
{
    opp = SOUTH;
}
else

    opp = NORTH
```

```
   // bean to empty spot case taking across hole to pot
   if (m_board.beans(endSide, endHole) == 1 and m_board.beans(opp, endHole) != 0 and
endSide== s)

      if (s == NORTH)


         m_board.moveToPot(NORTH, endHole, NORTH);
         m_board.moveToPot(SOUTH, endHole, NORTH);

      else


         m_board.moveToPot(NORTH, endHole, SOUTH);
         m_board.moveToPot(SOUTH, endHole, SOUTH);



   status(over, hasWinner, winner)

   // Check if the game is already over
   if (over)

      return false;


   if (endHole == 0 && over == false)// starts another move if bean to pot case



      if (endSide == NORTH)

         clear screen
         display()


         print "north extra move" end line
         bool on= false
         while (!on && !over)

            on = move(m_currentSide)
            status(over, hasWinner, winner)
            if (!over)

               print << "invalid move (NORTH) AAA" end line
```

```
            m_currentSide = SOUTH

        else

            clear screen
            display()

            print "south extra move" end line

            bool on = false;
            while (!on and !over)

                on = move(m_currentSide)

                status(over, hasWinner, winner)
                if (!over)

                    print  "invalid move (SOUTH) AAA" end line


            m_currentSide = NORTH


    return true


////////////////////////////////////////
void play()

    bool over = false
    bool hasWinner = 0
    Side winner

    while (!over)

        //check empty side case
        if (m_board.beansInPlay(NORTH) == 0)// rule empty side leads other side to take all of it

            for (int i = 1, i <= m_board holes, i++)

                m_board.moveToPot(NORTH, i, SOUTH)


        if (m_board.beansInPlay(SOUTH) == 0)
```

```
    for (int i = 1; i <= m_board.holes(); i++)

        m_board.moveToPot(SOUTH, i, NORTH)



clear screen
display()

if (m_currentSide == NORTH)

    print "north move" end line

    while (!move(m_currentSide) && !over)

        status(over, hasWinner, winner)
        if (!over)


            print "invalid move (NORTH)" end line




    m_currentSide = SOUTH;

else

    print "south move" end line
    while (!move(m_currentSide) and !over)

        status(over, hasWinner, winner);
        if (!over)

            print "invalid move (SOUTH)" end line;




    m_currentSide = NORTH;



    // rule empty side leads other side to take all of it
if (m_board.beansInPlay(NORTH) == 0 or m_board.beansInPlay(SOUTH) == 0)
```

```
        for (int i = 1; i <= m_board.holes(); i++)
            m_board.moveToPot(SOUTH, i, SOUTH)
        for (int i = 1; i <= m_board.holes(); i++)
            m_board.moveToPot(NORTH, i, NORTH)


    status(over, hasWinner, winner)

if (hasWinner)
    clear screen
    print "GAME OVER" end line
    display()
    if (winner == NORTH)
        print "NORTH: " + m_northPlayer->name() + " player wins" end line
    else
        print "SOUTH: " + m_northPlayer->name() + " player wins" end line

else
    print "tie game" end line
```

4) a note about any known bugs, serious inefficiencies, or notable problems you had.

The prompt for a human player to press enter periodically to continue is not working, it will only print the message but there is no way to press enter to continue in the game, but other than that the program functions fully.

a list of the test cases that would thoroughly test the functions. Be sure to indicate the purpose of the tests. Even if you do not correctly implement all the functions, you can still list test cases that would test them. Don't lose points by thinking "Well, I didn't implement this function, so I won't bother saying how I would have tested it if I *had* implemented it."

//board test cases

Skip opposite hole when sowing around the pot

case 1: south sow

```
Board b(3, 3);

b.setBeans(SOUTH, 3, 6);

Side s;
int e;

b.sow(SOUTH, 3, s, e);

assert(b.beans(NORTH, 0) == 0 && b.beans(SOUTH, 0) == 1);
```

case 2: north sow

```
Board b(3, 3);

b.setBeans(NORTH, 3, 6);

Side s;
int e;

b.sow(NORTH, 3, s, e);

assert(b.beans(NORTH, 0) == 1 && b.beans(SOUTH, 0) == 0);
```

sow spreads the beans numerically

case 1: south sow

```
Board b(3, 0);

b.setBeans(SOUTH, 1, 3);

Side s;
int e;

b.sow(SOUTH, 1, s, e);

assert(b.beans(SOUTH, 2) == 1 && b.beans(SOUTH, 3) == 1 && b.beans(SOUTH, 0) == 1 );
```

case 2: north sow

```
Board b(3, 0);

b.setBeans(NORTH, 3, 3);

Side s;
int e;

b.sow(SOUTH, 1, s, e);

assert(b.beans(NORTH, 2) == 1 && b.beans(NORTH, 1) == 1 && b.beans(NORTH, 0) == 1 );
```

GIVEN CASES

```
            Board b(3, 2);
            assert(b.holes() == 3  &&  b.totalBeans() == 12  &&
                        b.beans(SOUTH, POT) == 0  &&  b.beansInPlay(SOUTH) ==
6);
            b.setBeans(SOUTH, 1, 1);
            b.moveToPot(SOUTH, 2, SOUTH);
            assert(b.totalBeans() == 11  &&  b.beans(SOUTH, 1) == 1  &&
                        b.beans(SOUTH, 2) == 0  &&  b.beans(SOUTH, POT) == 2  &&
                        b.beansInPlay(SOUTH) == 3);
            Side es;
            int eh;
            b.sow(SOUTH, 3, es, eh);
            assert(es == NORTH  &&  eh == 3  &&  b.beans(SOUTH, 3) == 0  &&
                        b.beans(NORTH, 3) == 3  &&  b.beans(SOUTH, POT) == 3  &&
                        b.beansInPlay(SOUTH) == 1  &&  b.beansInPlay(NORTH) ==
7);
```

game test cases

Board in game should have a shallow copy of the board in the parameter

Game rules: enough to see the display case to check

//capture move case: choose move 1

```
Board b(3, 3);
Board b(3, 3);
   b.setBeans(SOUTH, 1, 1);
   b.setBeans(SOUTH, 2, 0);
   b.setBeans(SOUTH, 3, 1);

   Player* p1 = new HumanPlayer("bob");
   Player* p2 = new SmartPlayer("joe");

   Game g(b, p1, p2);
   g.move(SOUTH)
```

//empty side case: choose move 3

```
Board b(3, 3);
   b.setBeans(SOUTH, 1, 0);
   b.setBeans(SOUTH, 2, 0);
   b.setBeans(SOUTH, 3, 1);

   Player* p1 = new HumanPlayer("bob");
   Player* p2 = new SmartPlayer("joe");

   Game g(b, p1, p2);
   g.move(SOUTH)
```

//extra turn case: choose move 3

```
Board b(3, 3);
   b.setBeans(SOUTH, 1, 1);
   b.setBeans(SOUTH, 2, 0);
   b.setBeans(SOUTH, 3, 1);
```

```
    Player* p1 = new HumanPlayer("bob");
    Player* p2 = new SmartPlayer("joe");

    Game g(b, p1, p2);
    g.move(SOUTH)
```

Game end mechanic

//tie game

```
Board b(3, 0);

    Player* p1 = new HumanPlayer("bob");
    Player* p2 = new SmartPlayer("joe");

    Game g(b, p1, p2);
    bool over;
    bool hasWinner;
    Side winner;
    g.status(over, hasWinner, winner);
    assert(over == true && hasWinner == false);
```

//North wins game

```
Board b(3, 0);
    b.setBeans(NORTH,0, 1);

    Player* p1 = new HumanPlayer("bob");
    Player* p2 = new SmartPlayer("joe");

    Game g(b, p1, p2);
    bool over;
    bool hasWinner;
    Side winner;

    g.status(over, hasWinner, winner);
    assert(over == true && hasWinner == true && winner== NORTH);
```

//South wins game

```
Board b(3, 0);
    b.setBeans(SOUTH,0, 1);

    Player* p1 = new HumanPlayer("bob");
    Player* p2 = new SmartPlayer("joe");

    Game g(b, p1, p2);
    bool over;
    bool hasWinner;
    Side winner;

    g.status(over, hasWinner, winner);
    assert(over == true && hasWinner == true && winner== SOUTH);
```

//game not ended

```
Board b(3, 3);
```

```
Player* p1 = new HumanPlayer("bob");
Player* p2 = new SmartPlayer("joe");

Game g(b, p1, p2);
bool over;
bool hasWinner;
Side winner;

g.status(over, hasWinner, winner);
assert(over == false);
```

GIVEN CASES

```
HumanPlayer hp("Marge");
            assert(hp.name() == "Marge"  &&  hp.isInteractive());
            BadPlayer bp("Homer");
            assert(bp.name() == "Homer"  &&  !bp.isInteractive());
            SmartPlayer sp("Lisa");
            assert(sp.name() == "Lisa"  &&  !sp.isInteractive());
            Board b(3, 2);
            b.setBeans(SOUTH, 2, 0);
            cout << "=========" << endl;
            int n = hp.chooseMove(b, SOUTH);
            cout << "=========" << endl;
            assert(n == 1  ||  n == 3);
            n = bp.chooseMove(b, SOUTH);
            assert(n == 1  ||  n == 3);
            n = sp.chooseMove(b, SOUTH);
            assert(n == 1  ||  n == 3);
```