Project 2

Multiplayer Poker V2

https://github.com/KyleRiebeling/Poker_V2

CIS-17C 47065

Kyle Riebeling

5/30/2023

Introduction

Poker is a very common gambling game. Players will place bets to go to the "pot", which the winner earns after the round is over.

Each player pays a buy-in bet, then is dealt two cards. Based on what the two cards are and how the player feels, they can either fold and forfeit their bets for the round, raise the bets and require everyone else to do the same or give up, or call and just stay in the game and wait for the next round. This betting method is held after every card dealing round. The card dealing rounds are as follows: deal 2 cards to the players, play 3 cards on the table, place one more card on the table, and place one last card on the table.

Using the 5 cards on the table and the 2 cards in their hand, the players try to create the best hand of 5 that they can. The ranking of best hand to worst hand is as follows: one pair, two pairs, three of a kind, straight, flush, full house, and four of a kind. The player who had the highest ranking hand at the end of the last card deal wins the money in the pot. In the event of a tie in hand rankings, the winner is determined by the highest ranking card relative to what hand they have.

The functionality I have added in this version using recursions and binary trees include an optimized shuffle algorithm and a match history function that displays at the end of the game

Summary

Project size: about 1000 lines
Number of variables: about 26
Number of functions: 38

Techniques used:

Recursive sort: The random_shuffle function was replaced with a manual recursive shuffle function in order to shuffle the deck of cards. This allowed me to get rid of the Queue that held the shuffled deck of cards and let me use just the one deck of cards array.

AVL Tree: An AVL Tree was used to store the data from each round played in order to display at the end of the game. The key to each node was the round number. Each node held the winner's name, hand result, and the amount of money won in that round. When the game is over, an inorder traversal is used to display the results starting from round one.

Recursions: There are recursive calls all throughout the added functionality of the game. Most functions in the AVLTree class use recursive calls.

Graph: A Bidirectional Matrix Graph was used to track the hand results for each player after each round. The first three vertices correlated to the 3 possible players. Vertex 4 was reserved for error checking. The next 9 vertices reflected the possible poker hands. At the end of each round, an edge was either added to the new hand result for each player, or weight was added to the existing edge if the player already ended with that hand result. When displaying game stats at the end, the graph was analyzed to find the highest numbered vertex with an edge between each player vertex to display the highest hand each player received across all rounds. The graph then found the most weighted edge between each player vertex to display the most common hand the player ended with each round. In the case of a tie, the higher valued hand was chosen. Minimum spanning tree and shortest path algorithms were added into the class, but not used. They were added to show my ability to implement them. There was no purpose to use these algorithms in this simple project, so it felt wrong to use them here.

Pseudo Code

```
function main():
  numPlayers = 0
  tempC = empty character

  print "Welcome to the casino! How many people will be playing today? (2 or 3)"
  read numPlayers

  while numPlayers <= 0 OR numPlayers >= 4 OR cin.fail():
    print "Enter either 2 or 3: "
    clear input buffer
    read numPlayers

  pTable = new Table(numPlayers)

  print new line and "Welcome to the table!"
  pTable.printPlayers()

  while tempC != 'q':
    pTable.setTurn(1)
    for i in range 0 to 4:
      pTable.placeBets()
      pTable.startTurn()

    clear console screen
    print "Would you like to keep playing? Press 'q' to quit or anything else to play again: "
    read tempC

  clear console screen
  print "------Round History------"
  pTable.displayHistory()
```

```
 return 0
```

Major Game Loop Functions:

function startTurn():

```
  tempS: string
  currPlayer: integer
  currPlayer = 1

  if size of activePlayers is 1 and turn is less than 10:
    output "Everyone else folded,"
    initialize an iterator it for activePlayers
    set players[it->first] to it->second
    declareWinner(1)
    return

  if turn is 1:
    while player1Cards is not empty:
      remove the last element from player1Cards

    while player2Cards is not empty:
      remove the last element from player2Cards

    while player3Cards is not empty:
      remove the last element from player3Cards

    while dealerCards is not empty:
      remove the last element from dealerCards

    myDeck.dealHand(player1Cards, 2)
    myDeck.dealHand(player2Cards, 2)
    myDeck.dealHand(player3Cards, 2)

    initialize an iterator it for activePlayers
    while it has not reached the end of activePlayers:
      clear the system console
      output it->first + ", press any key and then enter to view your hand: "
      input tempS
      switch currPlayer:
        case 1:
          myDeck.viewHand(player1Cards)
          break
```

```
            case 2:
                myDeck.viewHand(player2Cards)
                break
            case 3:
                myDeck.viewHand(player3Cards)
                break
            otherwise:
                break

        increment currPlayer
        move to the next element in it

otherwise, if turn is 2:
    myDeck.dealHand(dealerCards, 3)  // Give the dealer 3 cards
    initialize an iterator it for activePlayers
    while it has not reached the end of activePlayers:
        clear the system console
        output it->first + ", press any key and then enter to view your hand with the cards on the table: "
        input tempS
        switch currPlayer:
            case 1:
                myDeck.viewHand(player1Cards, dealerCards)
                break
            case 2:
                myDeck.viewHand(player2Cards, dealerCards)
                break
            case 3:
                myDeck.viewHand(player3Cards, dealerCards)
                break
            otherwise:
                break

        increment currPlayer
        move to the next element in it

otherwise, if turn is 3:
    add myDeck.dealCard() to the front of dealerCards
    initialize an iterator it for activePlayers
    while it has not reached the end of activePlayers:
        clear the system console
        output it->first + ", press any key and then enter to view your hand with the cards on the table: "
        input tempS
        switch currPlayer:
           case 1:
              myDeck.viewHand(player1Cards, dealerCards)
              break
           case 2:
              myDeck.viewHand(player2Cards, dealerCards)
              break
           case 3:
              myDeck.viewHand(player3Cards, dealerCards)
              break
           otherwise:
              break

        increment currPlayer
        move to the next element in it

  otherwise, if turn is 4:
     add myDeck.dealCard() to the front of dealerCards
     initialize an iterator it for activePlayers
     while it has not reached the end of activePlayers:
        clear the system console
        output it->first + ", press any key and then enter to view your hand with the cards on the table: "
        input tempS
        switch currPlayer:
           case 1:
              myDeck.viewHand(player1Cards, dealerCards)
              break
           case 2:
              myDeck.viewHand(player2Cards, dealerCards)
              break
           case 3:
              myDeck.viewHand(player3Cards, dealerCards)
              break
           otherwise:
              break

        increment currPlayer
        move to the next element in it

  otherwise, if turn is 5:
     p1: pair of integers
     p2: pair of integers
```

```
        p3: pair of integers
        initialize p1 as (0, 0)
        initialize p2 as (0, 0)
        initialize p3 as (0, 0)

        initialize an iterator itr for activePlayers
        while itr has not reached the end of activePlayers:
            output a new line followed by itr->first + "'s hand result: "
            set players[itr->first] to itr->second
            switch currPlayer:
                case 1:
                    p1 = myDeck.evaluateHand(player1Cards, dealerCards)
                    break
                case 2:
                    p2 = myDeck.evaluateHand(player2Cards, dealerCards)
                    break
                case 3:
                    p3 = myDeck.evaluateHand(player3Cards, dealerCards)
                    break
                otherwise:
                    break

            increment currPlayer
            move to the next element in itr

        winner = findWinner(p1, p2, p3)
        declareWinner(winner)

    increment turn
```

function placeBets():

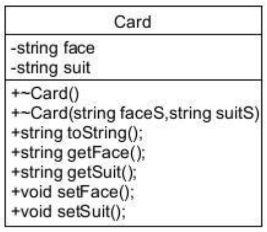
```
    currBet: integer
    tempC: character

    if turn is 1:
        increment round
        insert round into roundHist AVL tree
        currBet = 5
        output "Everyone places the buy-in bet of $5."
        clear activePlayers
    assign players to activePlayers
    initialize an iterator it for activePlayers
    while it has not reached the end of activePlayers:
        increase pot by currBet
        decrease it->second by currBet
        move to the next element in it

    output "The pot is up to $" + pot + "!"

otherwise, if turn is between 2 and 5 (inclusive):
    highBet = 0
    initialize an iterator it for activePlayers
    while it has not reached the end of activePlayers:
        if tempC is 'f':
            if size of activePlayers is 1:
                output "Everyone else folded, "
                declareWinner(1)
                return

            set it to the beginning of activePlayers

        tempC = ' '
        while tempC is not 'f' and tempC is not 'c' and tempC is not 'r':
            output it->first + ", make a choice: 'f' for fold, 'c' for call, or 'r' for raise: "
            clear cin
            ignore the remaining input in cin
            input tempC

        switch tempC:
            case 'f':
                players[it->first] = it->second
                output it->first + " is out of the game!"
                erase it->first from activePlayers
                break
            case 'c':
                break
            case 'r':
                if it->second is less than or equal to 0:
                    output "Unable to raise, calling instead. Press any key and enter to continue."
                    input tempC
                    break
        currPlay = it->first
        output "Enter amount to raise: "
        input highBet
        if it->second is greater than or equal to highBet:
            decrease it->second by highBet
            increase pot by highBet
        else:
            output "All in!"
            highBet = it->second
            it->second = 0
            increase pot by highBet

        raise(currPlay)
        output "New pot is $" + pot + "! Press any key and enter to continue: "
        input tempC
        return

    move to the next element in it
```

Major Variables

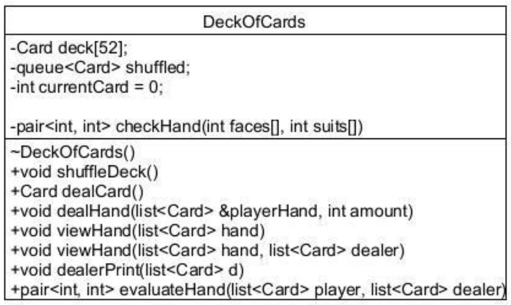
Type	Variable Name	Description	Location
Int	numPlayer	Amount of players	main()
Table	pTable	The table object for the program	main()
Int	turn	Tracks the current turn of the game	Class Table
map	players	Map that carries players names and their cash	Class Table
map	activePlayers	Map that carries temporary player data during gameplay to track who is active in each round	Class Table
int	pot	Total of bets placed per round	Class Table
Int	currentPlayer	Counter used to count each player per turn	Class Table

Type	Variable Name	Description	Location
list	player1/2/3/Cards dealerCards	Lists that hold each players cards per round	Class Table
int	highBet	Holds the amount raised that needs to be matched by each player if a player decided to raise	Class Table
int	currentCard	Counter for how many cards have been dealt	Class DeckOfCards
int[]	facesCounted	Counts how many of each face value are in a player's hand	DeckOfCards.evaluateHand()
int[]	suitsCounted	Counts how many of each suit are in a player's hand	DeckOfCards.evaluateHand()
string	face	Card's face value	Class Card
string	suit	Card's suit	Class Card
int	height	Node height in tree	Class Node
T	data	Key for nodes in tree	Class Node
String	winner	Winners name for match history	Class Node
int	HandVal	Winners hand	Class Node
int	winnings	Amount won that round	Class Node
Node<T> *	root	Base for the AVL tree	Class AVL
AVLTree<int>	roundHist	An AVL tree that holds the results of each round played	Class Table
Int **	graph	The 2d array that holds the graph data	Class BiMatrixGraph
int	numVerts	The number of vertices in the graph object	Class BiMatrixGraph

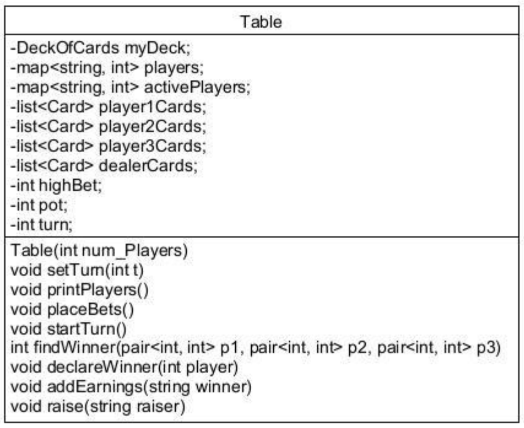
UML Diagrams



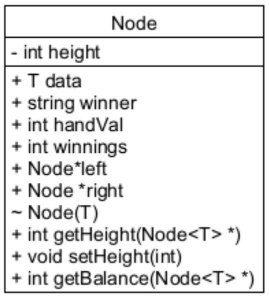
```
Card
-string face
-string suit
+~Card()
+~Card(string faceS,string suitS)
+string toString();
+string getFace();
+string getSuit();
+void setFace();
+void setSuit();
```



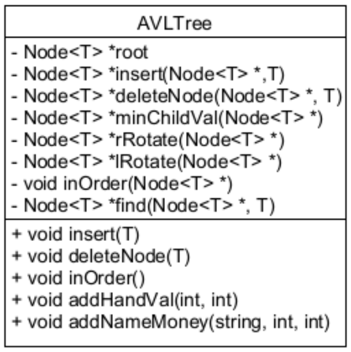
```
DeckOfCards
-Card deck[52];
-queue<Card> shuffled;
-int currentCard = 0;

-pair<int, int> checkHand(int faces[], int suits[])
~DeckOfCards()
+void shuffleDeck()
+Card dealCard()
+void dealHand(list<Card> &playerHand, int amount)
+void viewHand(list<Card> hand)
+void viewHand(list<Card> hand, list<Card> dealer)
+void dealerPrint(list<Card> d)
+pair<int, int> evaluateHand(list<Card> player, list<Card> dealer)
```



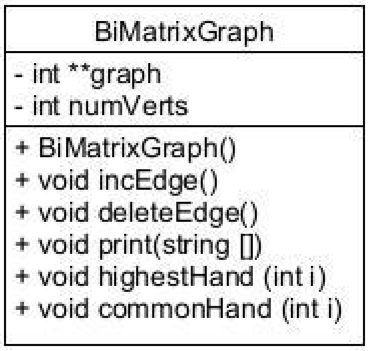
```
Table
-DeckOfCards myDeck;
-map<string, int> players;
-map<string, int> activePlayers;
-list<Card> player1Cards;
-list<Card> player2Cards;
-list<Card> player3Cards;
-list<Card> dealerCards;
-int highBet;
-int pot;
-int turn;
Table(int num_Players)
void setTurn(int t)
void printPlayers()
void placeBets()
void startTurn()
int findWinner(pair<int, int> p1, pair<int, int> p2, pair<int, int> p3)
void declareWinner(int player)
void addEarnings(string winner)
void raise(string raiser)
```



```
Node
- int height
+ T data
+ string winner
+ int handVal
+ int winnings
+ Node*left
+ Node *right
~ Node(T)
+ int getHeight(Node<T> *)
+ void setHeight(int)
+ int getBalance(Node<T> *)
```



```
AVLTree
- Node<T> *root
- Node<T> *insert(Node<T> *,T)
- Node<T> *deleteNode(Node<T> *, T)
- Node<T> *minChildVal(Node<T> *)
- Node<T> *rRotate(Node<T> *)
- Node<T> *lRotate(Node<T> *)
- void inOrder(Node<T> *)
- Node<T> *find(Node<T> *, T)
+ void insert(T)
+ void deleteNode(T)
+ void inOrder()
+ void addHandVal(int, int)
+ void addNameMoney(string, int, int)
```



```
BiMatrixGraph
- int **graph
- int numVerts
+ BiMatrixGraph()
+ void incEdge()
+ void deleteEdge()
+ void print(string [])
+ void highestHand (int i)
+ void commonHand (int i)
```