**Blackjack**

**Introduction**

I coded blackjack for my project due to it was a WIP that was for my 17B class this semester. When I originally created it, most of the code ran on vectors. I decided to retrofit it so that it’d run on queues, maps, sets and etc. or other containers. I spent about a week retrofitting the code so it’d work with the containers asked. There’s approximately 1200 lines of code in the project. It contains 7 classes which are Player, Login, Deck, Card, Hand, Bet, and Table. The login allows the user to create their own ID to save their score/money. Player is the unique ID. Cards form a deck, which is used in table in order to play the game. Hand handles the game logic for what a hand can do, and table is what brings it all together for a menu. The code can be found at [https://github.com/albuut/Blackjack17c](https://github.com/albuut/Blackjack17c%20).

**Approaches**

How I approached the project was I started from a base, like player and card. The player will be used by login, so most of the variables and functions will be to modulate the player. I then created cards, which I know would be used by a deck and hand since they’ll hold these values separately with their own logic. Afterwards I created a bet class in order to track my bets. All of these will go into table and allow the game to play out with it’s rules.

**Game Rules**

The goal of blackjack is to out-score the dealer’s hand without going over the value 21. When you first play, you enter with an account, afterwards you choose a value $100-$1000 for the size of your bets. Then, you would have chosen how many hands you would want to play, 1 through 5. You would then be dealt two cards for each hand. You’ll be given an option to double down, which is to double your current bet and only draw one more card, or you could choose to hit which would add another card to your hand. The values of each card are the same as their number shown, Ace has a value of 1 or 11 and the face cards have a value of 10. If you are to get blackjack, 21 on draw, then you automatically win that hand with a pay out of 3:2. The dealer also draws two cards in the beginning, but one is hidden from you. If the card that’s revealed is an ace. It gives you the option to do even money or insurance which would essentially, you’re gambling if the dealer has a blackjack. You keep hitting your hands until you decide to stop, when you finish hitting all your hands. The dealer will reveal their card and deal as many cards as they can until they reach the value 17. Once they hit a value over 17 they’ll stop. At that point there will be a showdown between your hands, if your hand value is larger than the dealers without going over 21. You win your bets.

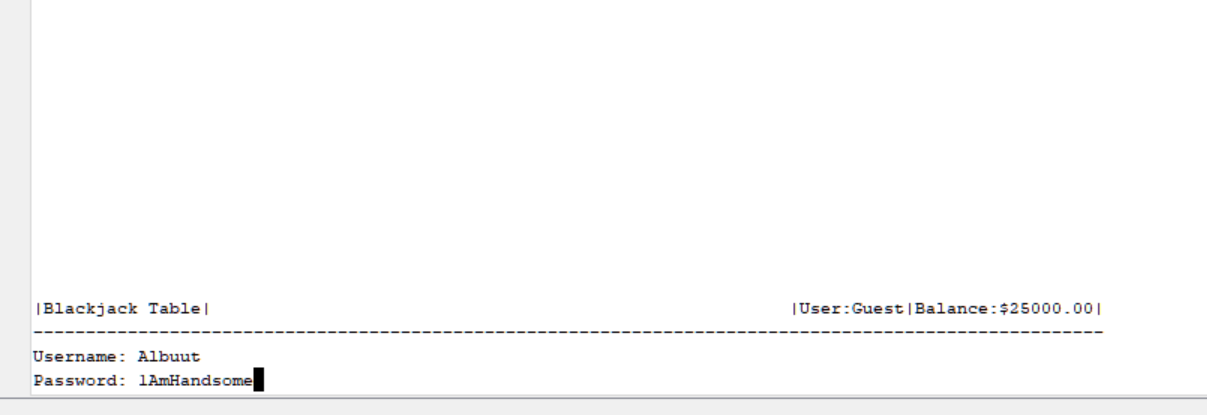
**Descriptions of Code**

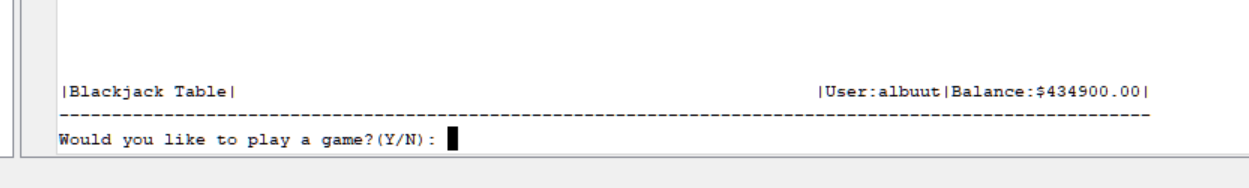
The code is separated into 7 classes and a main. The 7 classes are Player, Login, Deck, Hand, Table, Bet, and Card. Card is what contains the point values and the suit of each individual card. Deck and hand are classes which hold the cards. The purpose of deck is to shuffle them and distribute the cards in a random manner, Hand holds cards that the player would use and it has different logic steps in order to evaluate what action you can do, such as hitting, stand, double down, insurance, or even money. It also has the ability to calculate your hand total. Bets are the values of which you decide to bet. It also determines how much you win or lose per hand, for example if you hit blackjack, you get a higher ratio of winnings rather than the 1:1. You get 3:2. It essentially holds the checks if you won a bet or if you tied. Player stores your username and password. It also stores how much money you have won. It also internally stores how many hands you’ve played and won. This could be useful for later on if you wanted to have a state sheet or if you needed to detect cheaters due to observing abnormalities in a user. The table is essentially what runs the blackjack logic. It passes allows you to pass the cards drawn from your deck into hands, it checks if your hand wins compared to the dealer, it also is the logic which allows your dealer to play cards. Afterwards it’ll have a run through in order to observe which hands you won and properly divvy up how much money you’ve won or lost.

Newly implemented features are that there’s now a % chance indicator to help the user know what their percent chance of being able to safely hit would be, and the % chance if it’d be 21. Passwords are now saved and hashed in order to produce a safer password. There’s also a new way to allow them the option to pick a card after deciding not to double down

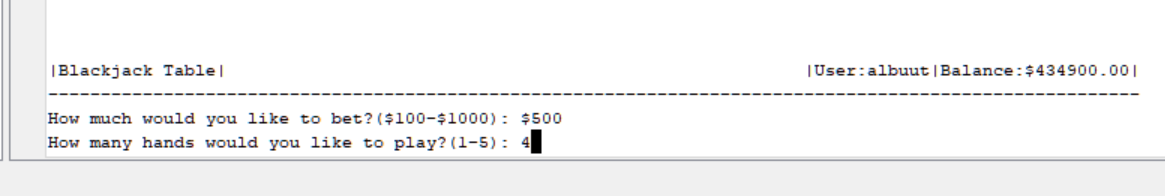
**Sample Input Output**

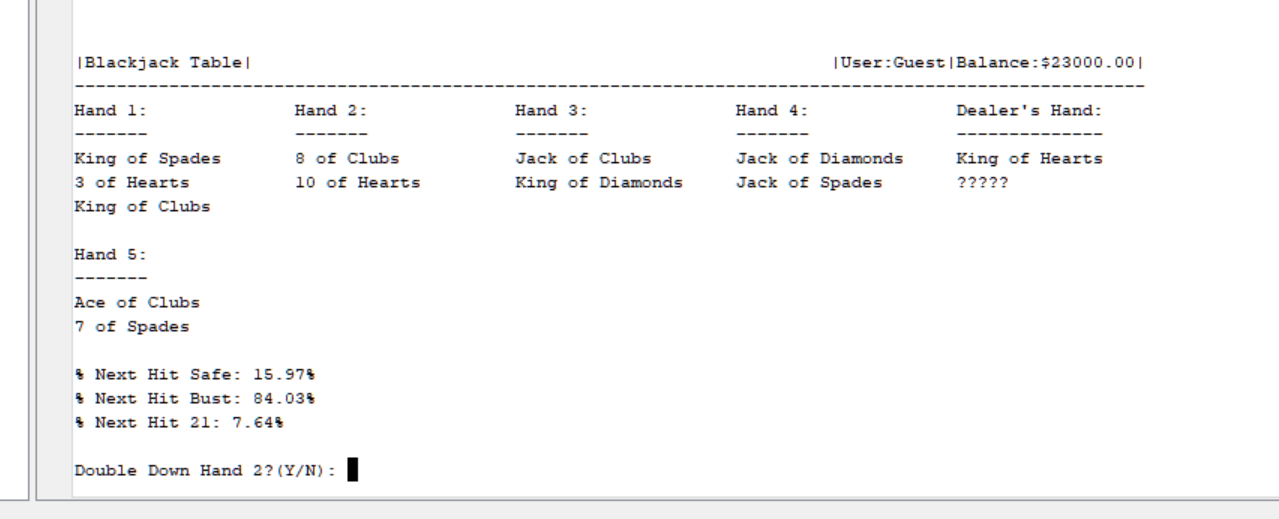
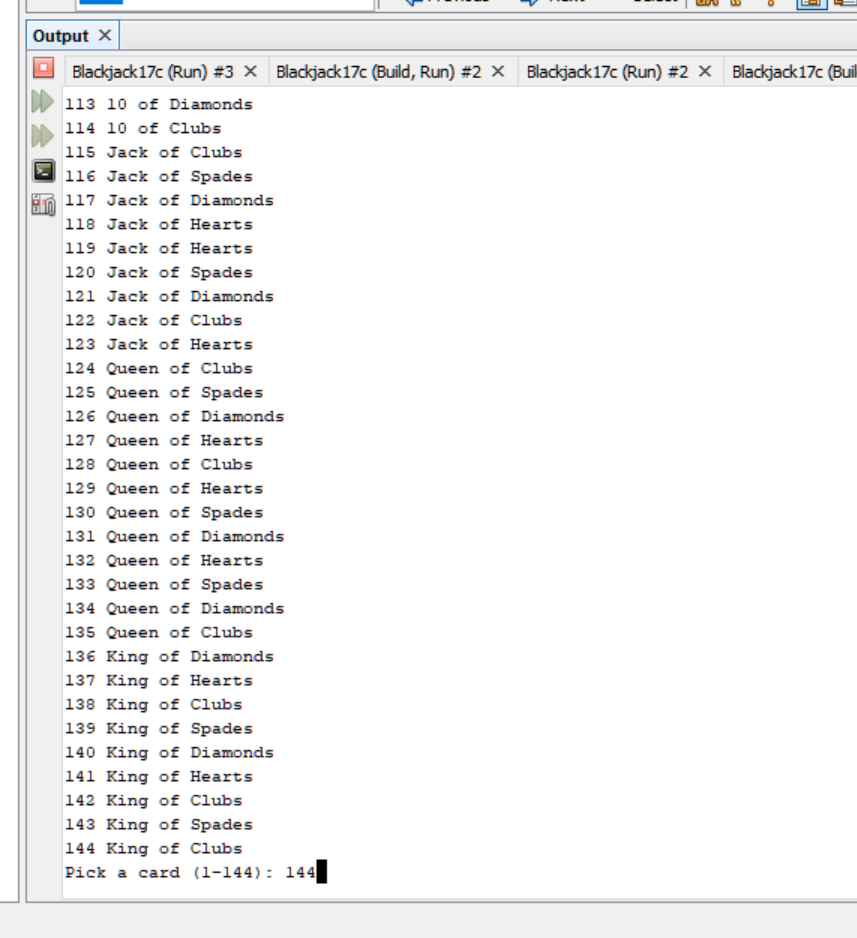
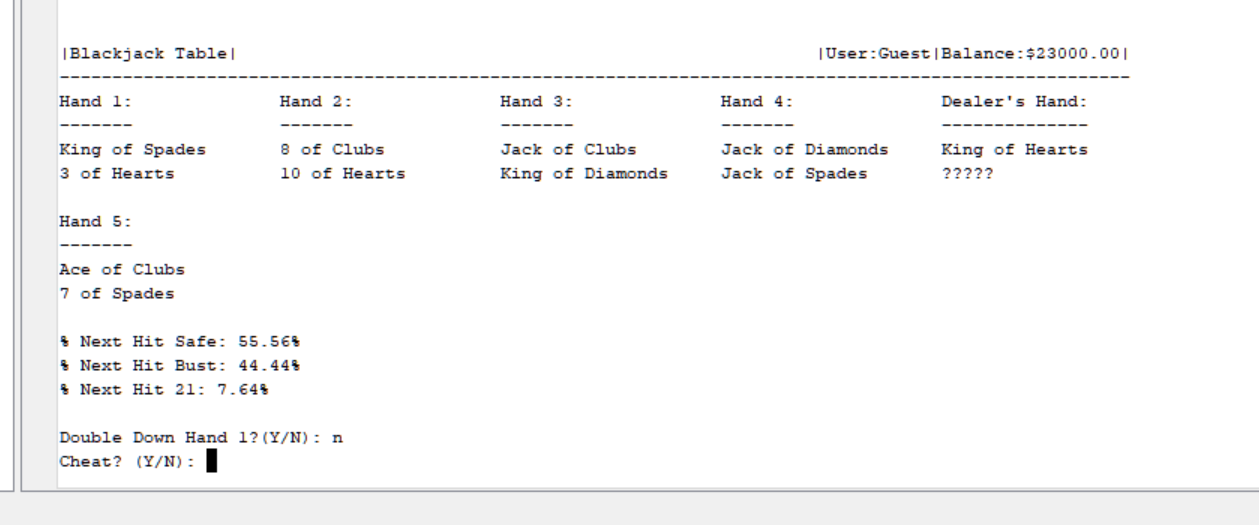
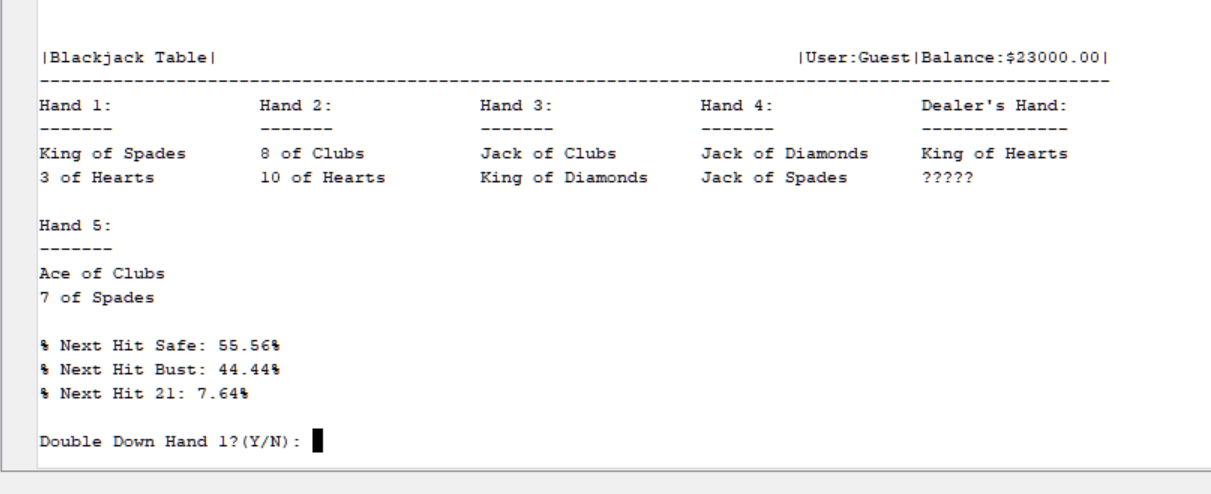


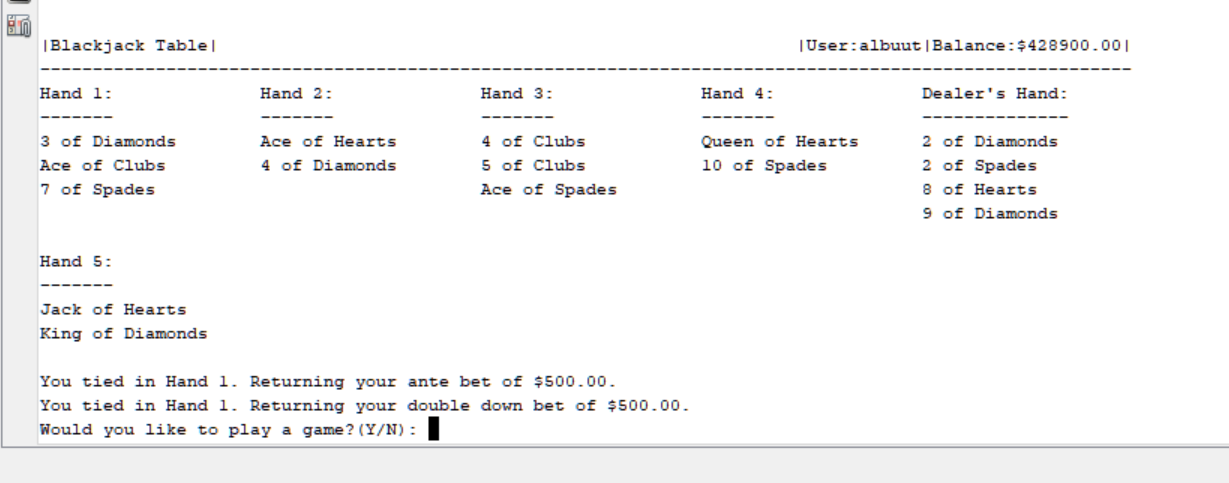






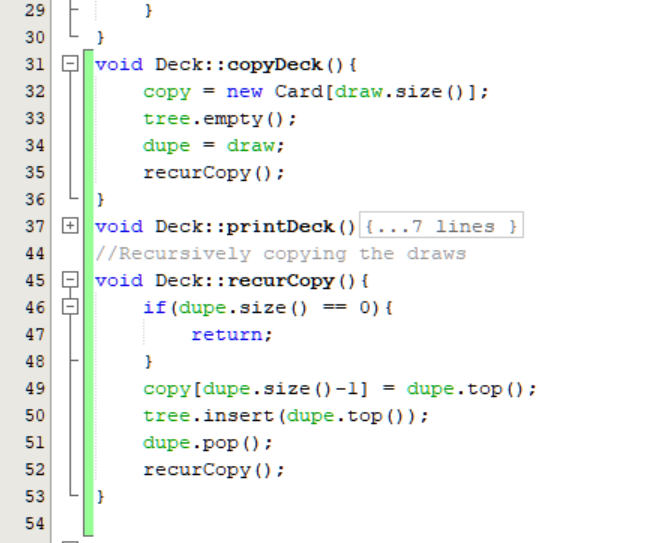






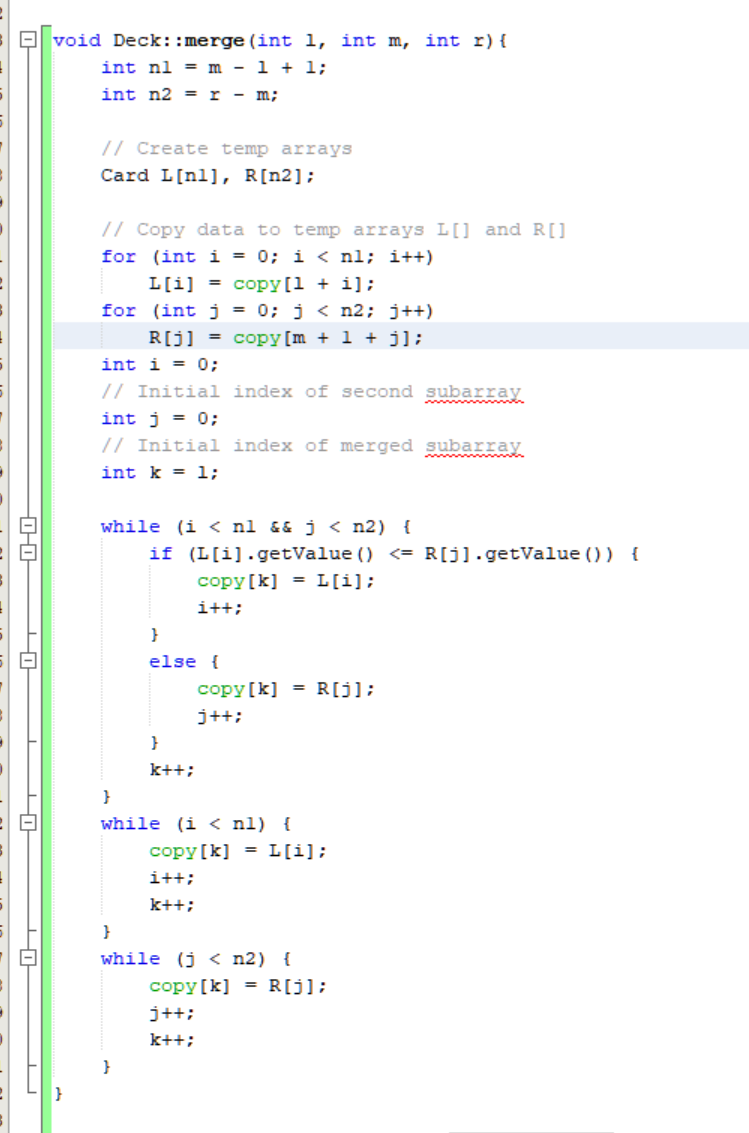
Checklist

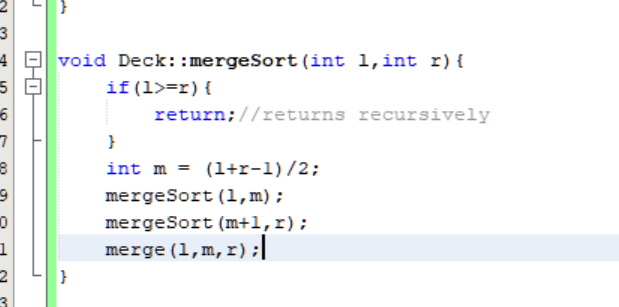
**Recursions**



For recursions, I decided to recursively fill an array and also input the values into my binary search tree to do data acquisition. This ends up leading into the manner in which I could easily and quickly find what’s in the deck. This is required due to I’m using a stack and I want to be able to access certain parts of the array to return a certain card.

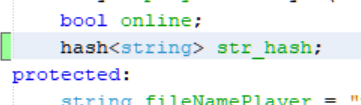
**Recursive Sorts**

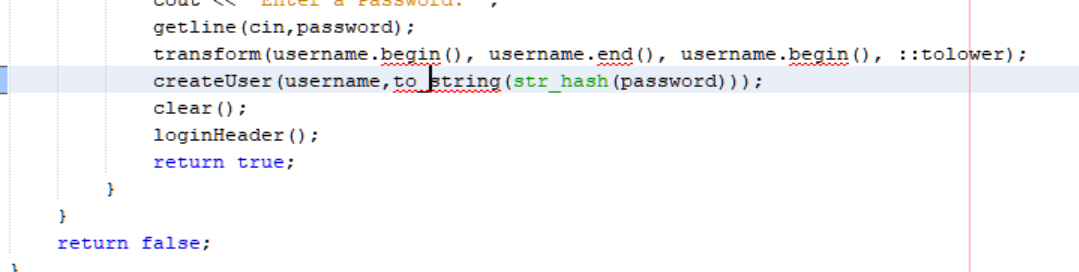


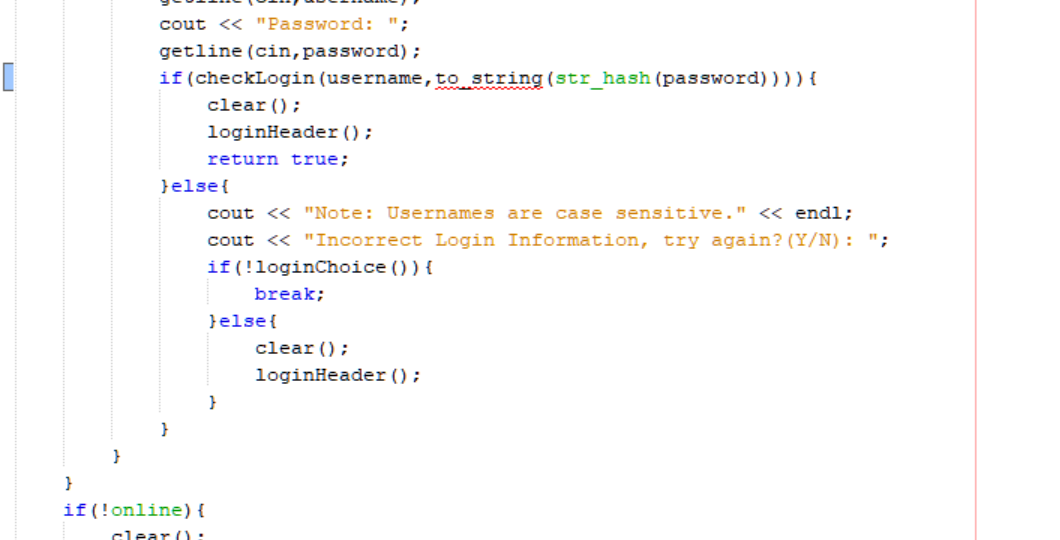


For my recursive sort, I decided to use a merge sort in order to sort the cards that are left inside of the deck. This is used to give the user an easier time to look. This recursively sorts all of my cards by their value. What could probably be improved is to also do a second case which also checks for card suit and order it that way also.

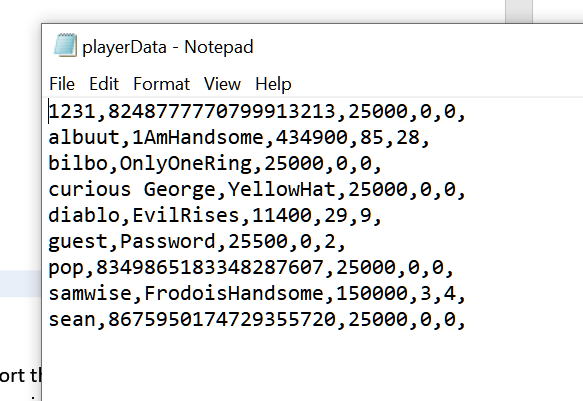
**Hashing**





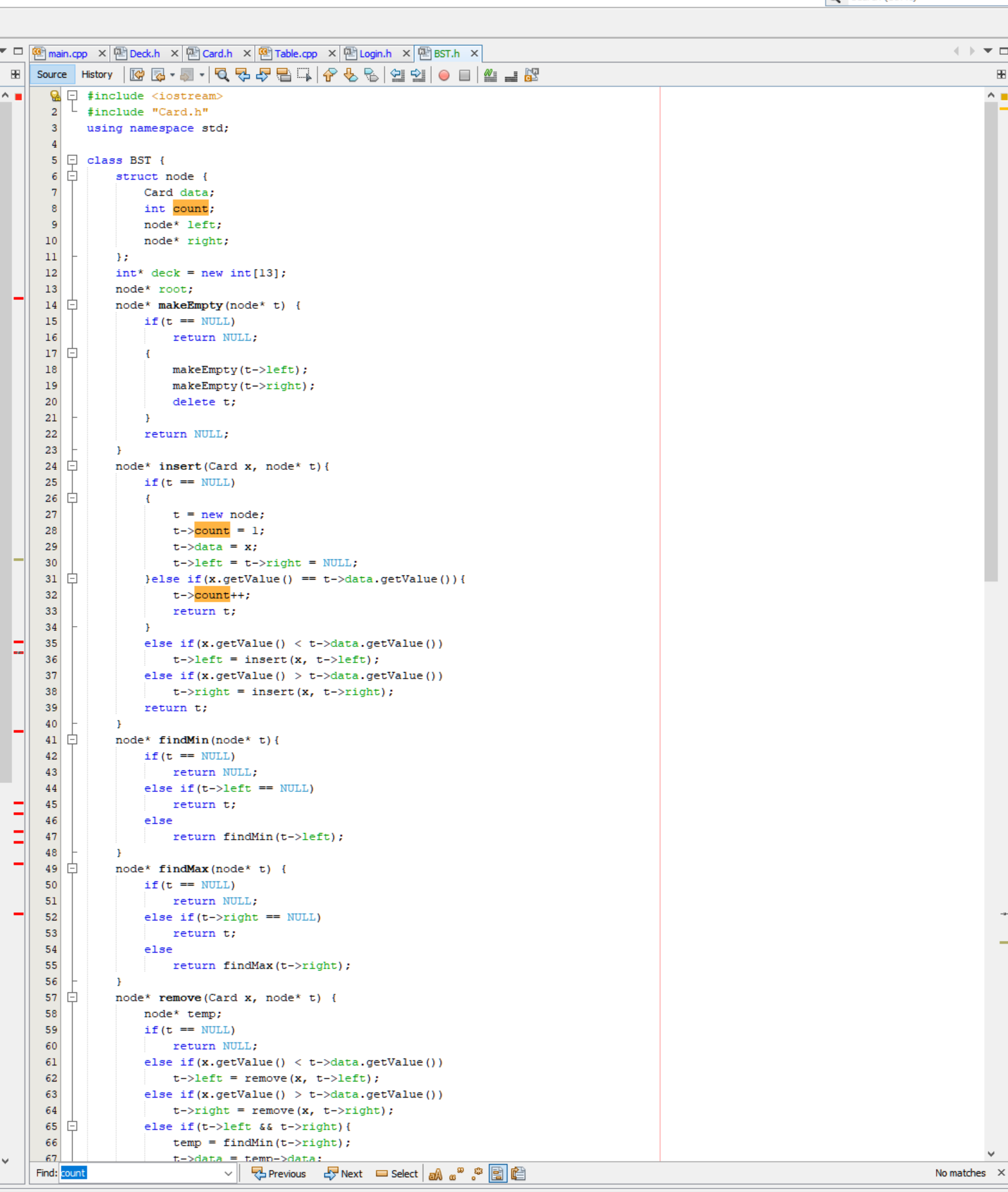


For hashing, I used the built in STL function to hash the input into some digit. This hash is then used to be stored when you create a user and when you have to validate your password. This is a way in which you have a more secure password in use.



If you observe the latest entry. You can see that when it’s saved to a file now. It’s saved as a series of digits now rather than what the intended password is. You can check this by creating a user and logging back in after you exit the program .

**Tree**



I used a binary search tree in order to store my values to count how many I have and have it so that they’re all stored in an orderly manner. THE BST will automatically sort them in the correct order and be able to transverse to collect all this data. Due to there being so many cards with the possibility of being a duplicate value. I have it so that it just counts how many extra entries there are if an extra card is added. This could probably be expanded by adding extra conditionals where we could have the cards sorted by value and then suit.

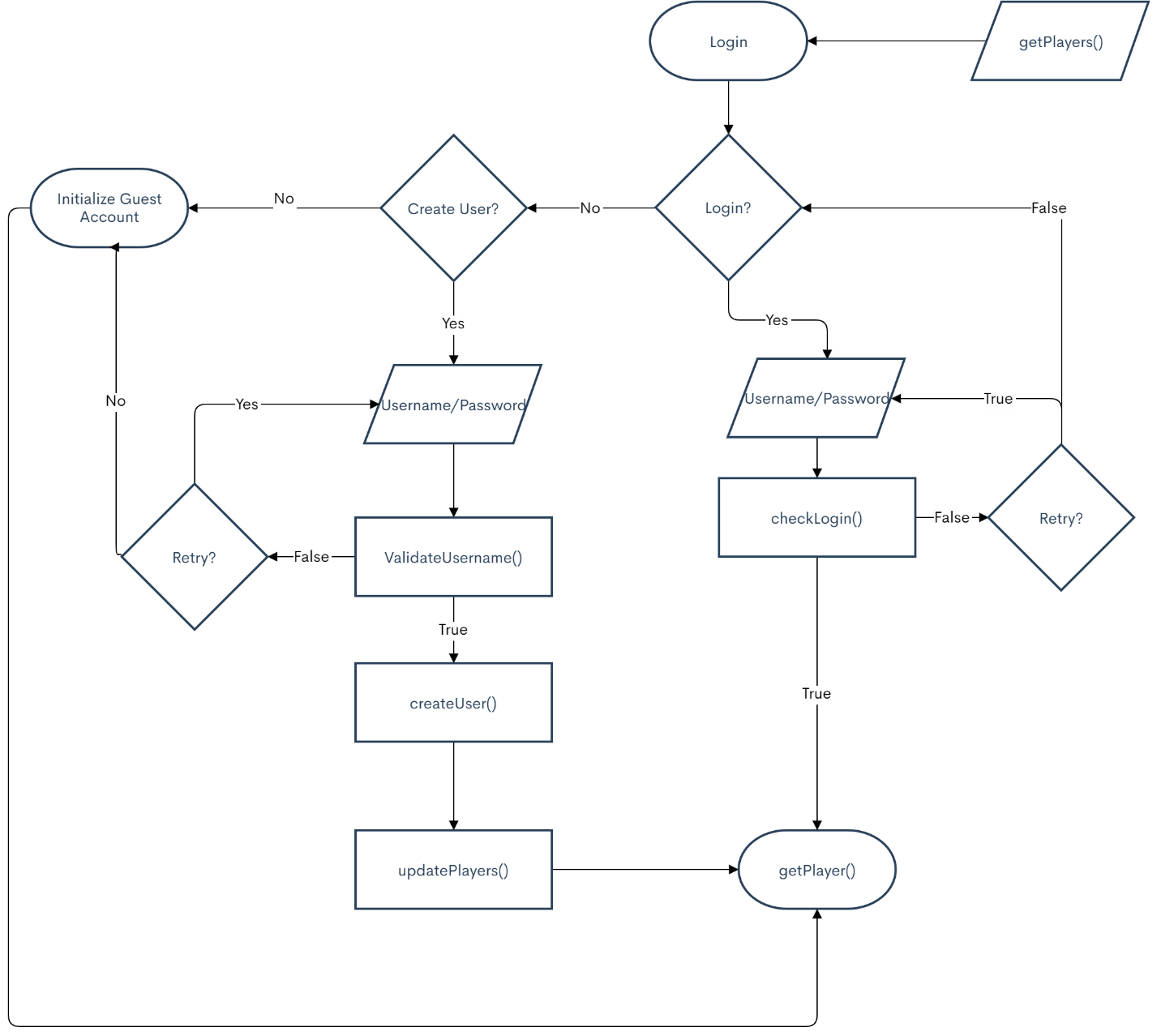
**Graphs**

No graphs for me. My brain is too small to come up with a solution in which using a graph in a blackjack game would make sense to me. There’s no transversals or links that I could think about where there is a requirement to have a direction for my nodes.

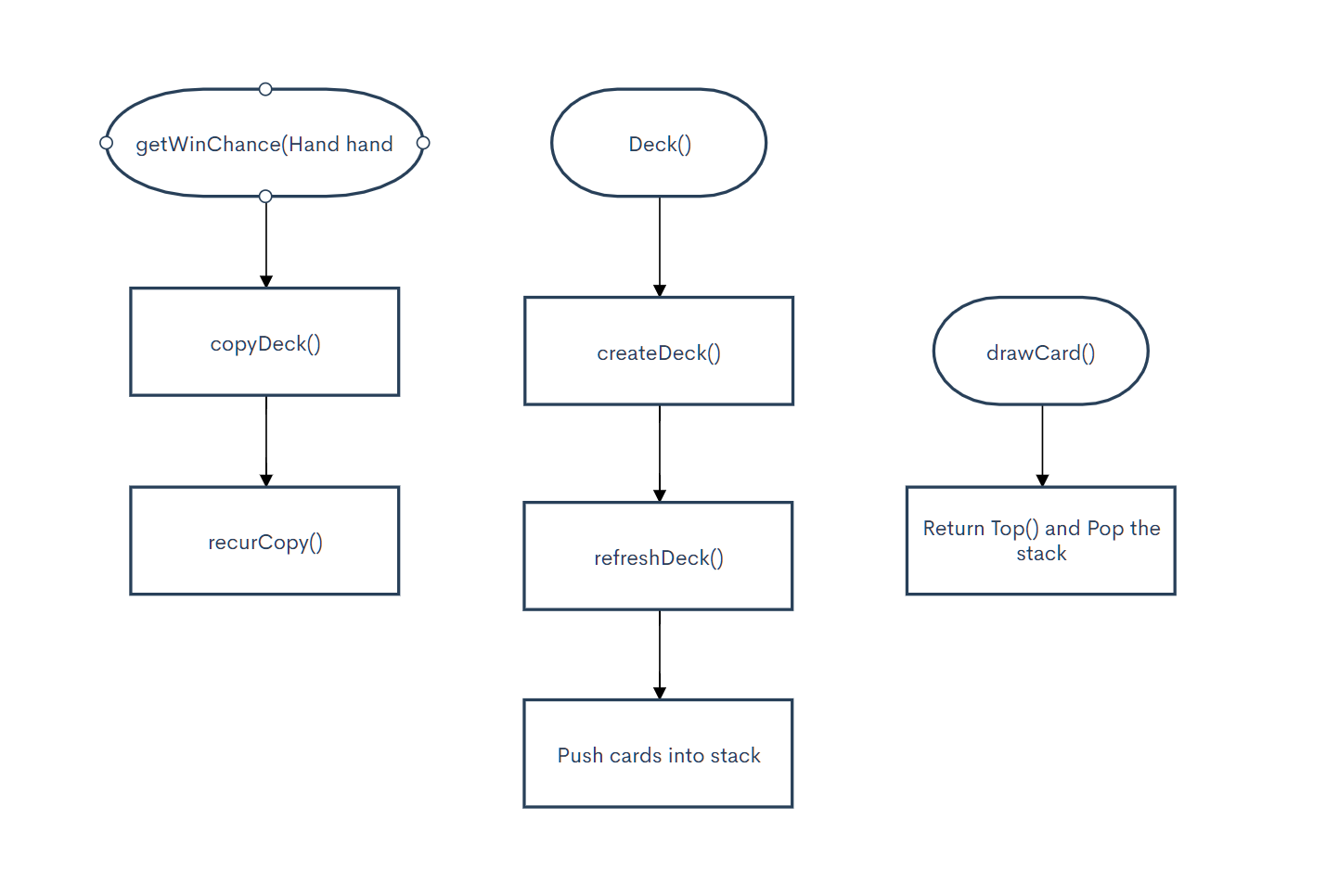
Code Documentation:

Flowcharts

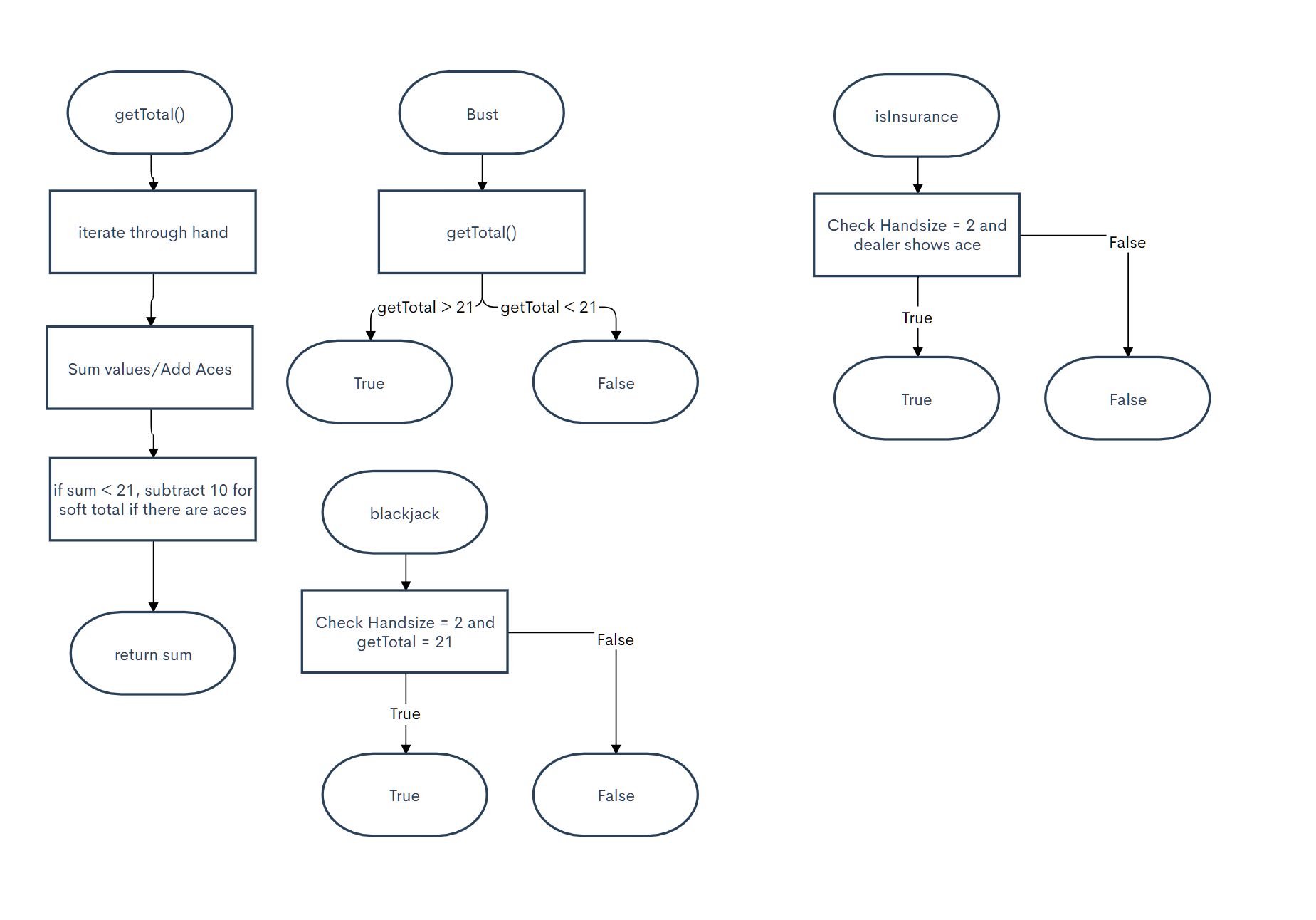
**Login.h**



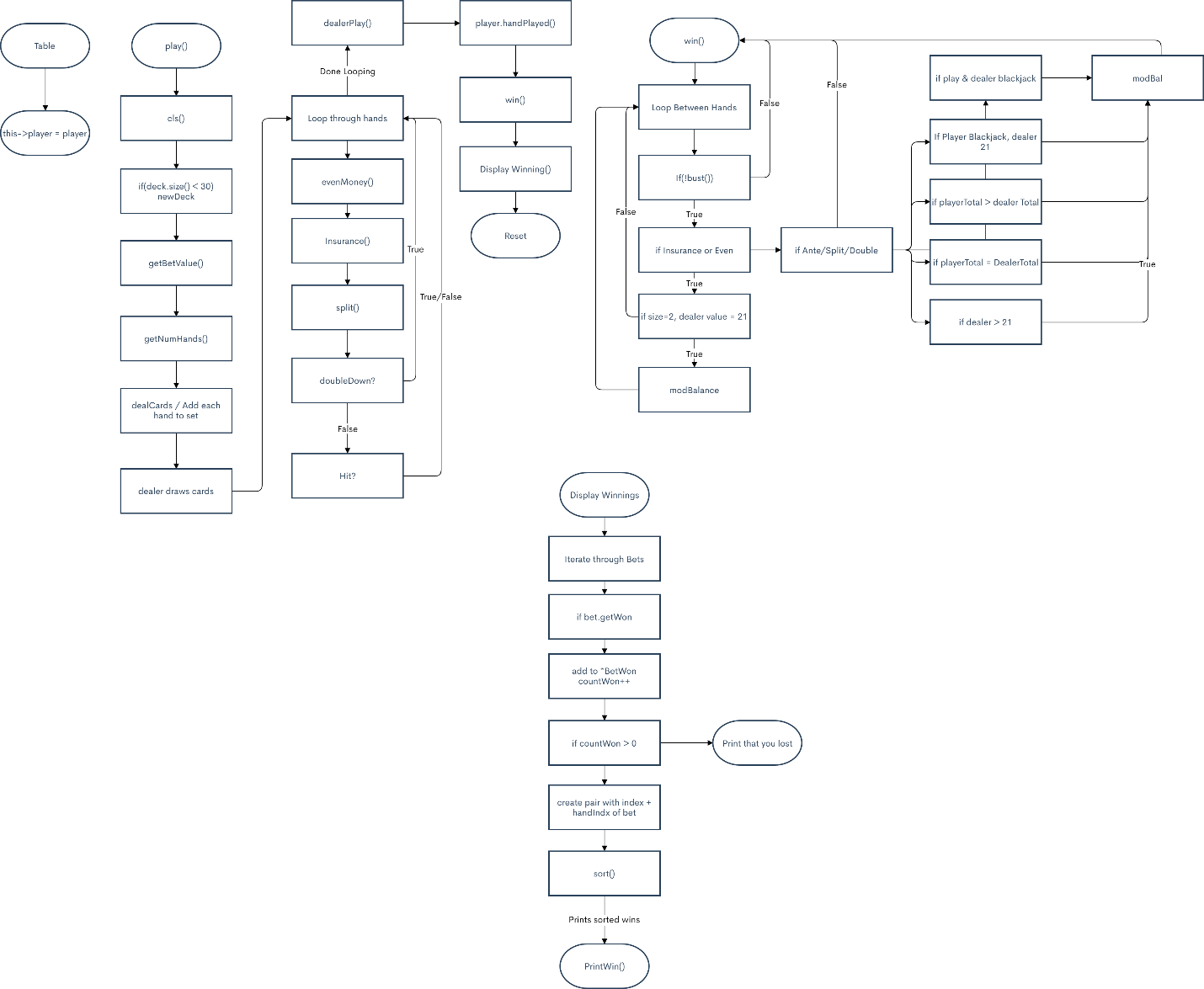
**Deck.h**

****

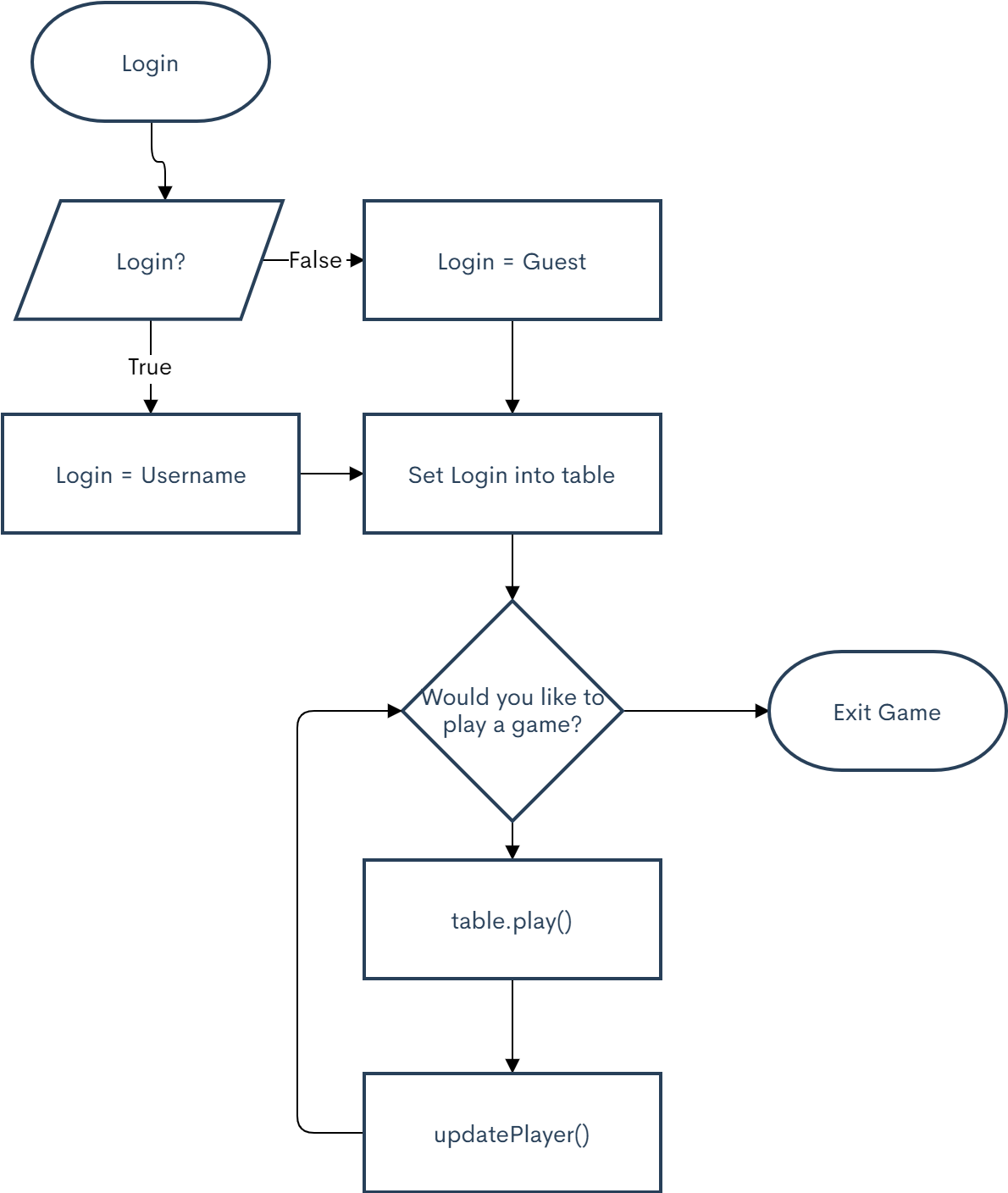
**Hand**

****

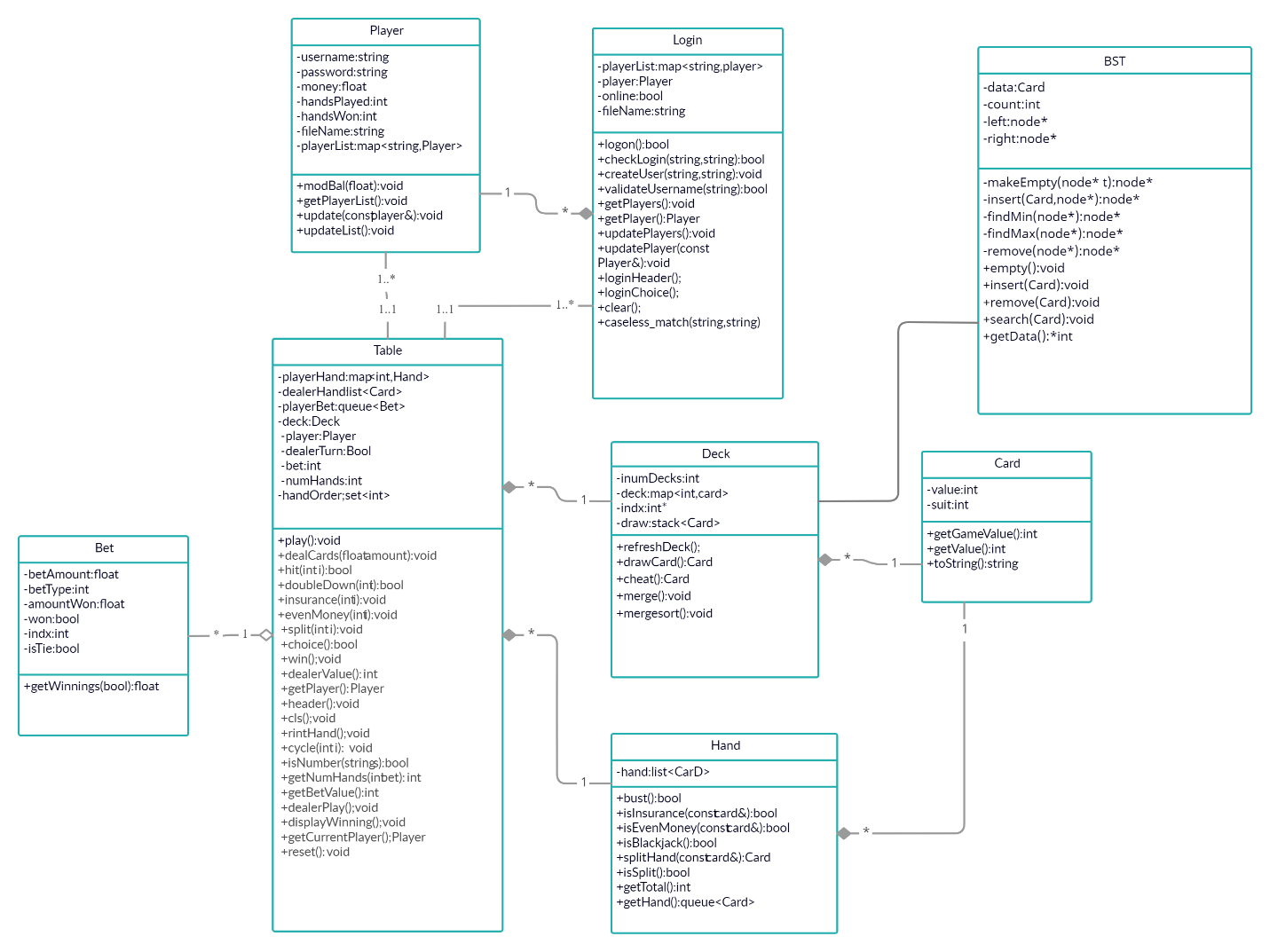
**Table**



**Main**



**UML**



**Psuedo Code:**

**Card->Deck**

**Card->Hand**

**Player->Login-**

**Player->Table**

**Login->Table**

**Bet->Table**

**Deck->Table**

**Hand->table**

**Table**

**Play**

**Create a deck**

**Create hands**

**Put Hand into container**

**Get amount for bet**

**Put bet into container**

**Play through Hands**

**Ask DoubleDown**

**Ask Cheat**

**Ask Hit/Insurance/Even Money**

**Make more bets depending on option**

**Dealer Plays**

**Hit til 17**

**Check Bets**

**If player = win**

**Bet = won**

**Print Winnings**

**Reset Board**

**Play;**

**Card = Value/Suit**

**Deck = container of card,deals cards**

**Hand = container of card with logic**

**Bet = holds bet amount, type, and if won**

**Login = create or login user**

**Player = amount won, hands played, username, password**