Program design:

This program’s goals are to act as a investment bank, the client can open accounts, withdraw funds, deposit funds, transfer funds, or ask for the transactional history to be printed. The client is identified by their unique 4 digit id’s. A fifth digit can be added to pull up their unique investment account. And 5 types of transactions can be performed; Open, Deposit, Withdraw, Transfer and display history. The third phase, you would print out each client account and the amount held in each specific account. You are expected to store client information in binary trees.

JollyBanker.py

Variables: (all 10 funds, stored as floats), client\_account(int), transactions(Queue), client\_account\_balance(float), client\_name(str), account\_list(Binary tree)

Methods: transaction(operation(char), account\_number(int), money\_amount(int)), transaction(operation(char), name(str), account\_number(int)), transaction(operation(char), account\_number(int)), transaction(operation(char), money\_amount(int)), transaction(operation, origin\_account(int), money\_amount(int), destination\_account(int)), open(name(str), id(int)), Deposit(account\_number(int), money\_amount(int)), withdraw(account\_number(int), money\_amount(int)), transfer(origin\_account(int), money\_amount(int), destination\_account(int)), history(account\_number(int))

Diagram

Description automatically generatedA pink paper with writing on it

Description automatically generated with low confidence

4. How classes would interact with each other

The orders from the in file would be read, and be putted into a queue, the class that would process the order would then be called. Each order would be process through in a FIFO manner, and read through the queue. After all of the orders are read, it would print out all accounts and the balances.

5. program flow:

Class reader:

Transaction\_list = []

For line in file:

Transaction\_list.append(line)

Banker(Transaction\_list)

Class Node:

Def \_\_init\_\_(self, data):  
 self.left = None

Self.right = None

Self.Data = data

Class binaryTree:

def \_\_init\_\_(self):

self.root = None

def addNode(self,data):

return Node(data)

def insert(self,root,data):

if(root == None):

root = self.addNode(data)

else:

if(data <= root.data):

root.left = self.insert(root.left,data)

root.right = self.insert(root.right,data)

return root

Class account:

Def \_\_init\_\_(self, name, number):

Self.name = name

Self.number = number

Def total\_val(self):

Add up all 10 accounts and return

Def deposit(self, account, amount):

Divide string and put in the proper amount

Def withdraw(self, account, amount):

Divide string and put in the proper amount

If insufficient funds, move funds from other accounts

Def transfer(self, origin, amount, target):

Identify origin account

Reduce and add

Increment target account

Def history(self, account):

Identify account number and break the integer

print

Class banker:

Def \_\_init\_\_(self, Transaction\_list):

List\_of\_accounts = []

While Transaction\_list != null:

Transact = Traction\_list.dequeue

(Divide string)

Transaction(transact)

While list\_of\_accounts != empty:

Print(account number + “ ” account.total\_value)

Def Transaction(self, char(order), str(name), int(acct\_num))

Data\_node = acct\_num  
 self.account\_tree.insert(Data\_node)

Add account to account list

Def Transaction(self, char(order), int(acct\_num), int(amount))

Break acct\_num to identify account

If order == ‘W’:

Call withdraw on that account

If order == ‘D’:

Call deposit on that account

Def Transaction(self, char(order), acct\_num):

Break string

Call history on that account