# Weekly Assessment 6

Instructions:

1. Sign the Honor Pledge.
2. In addition to your own notes and books, online resources can also be used but you have to provide reference(s).

Honesty Pledge:

I pledge on my honor that I have not given or received any unauthorized assistance on this exam/assignment.

Indicate your Name / Date: Kyle Matthew M. Cayanan / April 25, 2023

---

1. [20 pts] An airport is developing a computer simulation of air-traffic control that handles events such as landings and takeoffs. Each event has a time stamp that denotes the time when the event will occur. The simulation program needs to efficiently perform the following two fundamental operations:

- Insert an event with a given time stamp (that is, add a future event).
- Extract the event with smallest time stamp (that is, determine the next event to process).

Which data structure should be used for the above operations? Why?

## Answer:Min-Heap Data Structure

Reason: Min-Heap is chosen because "Heap" organizes data in numerical order. Because of its numerical nature, it is an efficient choice for "inserting a given time stamp". According to the lecture, inserting

For Min-Heap specifically, the smallest value is always in the root node. Therefore, searching for "the event with smallest time stamp" is efficient in this data structure.

2. [20 pts] What does each `remove_min` call return within the following sequence of priority queue ADT methods:

add(5,A), add(4,B), add(7,F), add(1,D), remove_min(), add(3,J), add(6,L), remove_min(), remove_min(), add(8,G), remove_min(), add(2,H), remove_min(), remove_min()?

## Answer:

Solution: - Interpretation: remove_min() refers to the removal of the lowest priority item

In order: - 5,A - 4,B - 1,D - 7,F - 8,G - 2,H

3. [60 pts] An online computer system for trading stocks needs to process orders of the form "buy 100 shares at $\$x$ each" or "sell 100 shares at $\$y$ each." A buy order for $\$x$ can only be processed if there is an existing sell order with price $\$y$ such that $y \leq x$. Likewise, a sell order for $\$y$ can only be processed if there is an existing buy order with price $\$x$ such that $y \leq x$. If a buy or sell order is entered but cannot be processed, it must wait for a future order that allows it to be processed.

a. Describe a scheme that allows buy and sell orders to be entered in $O(\log n)$ time, independent of whether or not they can be immediately processed.

## Answer:

I will implement insertion using the Heap Priority Data Structure. According to the lecture, insertion in this data type takes log(n) time.

Specifics: I will have two separate heaps for Buy Orders and Sell Orders. The function will no longer need to perform identification to distinguish orders. It can simply access the heap with the correct name (i.e. buy_holder, sell_holder).

Finally, the actual orders to be pushed. When pushing an order into a heap, I will push a touple where the "priority" is the order price, and the order details is the second element.

Example: heappush(buy_holder,(60,buy_order)) where priority=$60, and buy_order contains the quantity to be bought along with any other details necessary for the entire class to function properly

b. Write a program that can process a sequence of stock buy and sell orders as described above. Use the Python heapq module.

1. The class name must be `StockExchange`.
2. Implement the following methods: `buy()`, `sell()`, `process_orders()`.
3. Both `buy()` and `sell()` methods must call `process_orders()`.

Expected program usage:

```
trading = StockExchange()
trading.buy(100, 30)
trading.sell(50, 20) #Output: 50 shares sold at $20
trading.buy(50, 25)
trading.sell(100, 25) #Output: 50 shares sold at $25
```

**Provide your code and attach a recorded audio explanation (You can use online tools like http://veed.io to record your audio).**

Some notes:

We are taking the perspective of a stock exchange.

The orders will be served based on priority of the bid prices. Such that the a buy order with bid price x will be processed when there is sell order with offer price y such that y<=x. Orders with the highest bid price get priority in being matched to sell orders.

```
Buy 100 shares at $20
Buy 100 shares at $15
Sell 100 shares at $10
- The "Buy 100 shares at $20" will be matched with "Sell 100 shares at $10" first.

Sell 100 shares at $15
Sell 100 shares at $10
Buy 100 shares at $20
- The "Sell 100 shares at $10" will be matched with "Buy 100 shares at $20" first.
```

```
# YOUR CODE HERE
#import the heapq module
import heapq

#create the class
class StockExchange():
    def __init__(self):
        #first initialize the heaps that will contain the orders
        self.heap_buy=[] #heap that will contain buy order
        self.heap_sell=[] #heap that will contain sell order

        #initialize the heaps that will be used to iterate through all the orders
        self.buy_holder=[]
        self.sell_holder=[]



    #push the buy order onto the buy heap
    def buy(self,quantity,price):
        heapq.heappush(self.heap_buy,(price,quantity))
        self.process_orders()

    #push the sell order onto the sell heap
    def sell(self,quantity,price):
        heapq.heappush(self.heap_sell,(price,quantity))
        self.process_orders()

    #process all possible orders
    def process_orders(self):
        #initialize holders

        #holds the touples
        sell=None
        buy=None
        #enables the while function
        finish=False

        #merge the matching heaps at the start
        for i in self.sell_holder:
            #merge the sell heap
            heapq.heappush(self.heap_sell,i)
        for j in self.buy_holder:
            #merge the buy heap
            heapq.heappush(self.heap_buy,j)

        #reverse the buy_heap in order to make it from largest to smallest
        self.heap_buy.reverse()

        #if buy or sell is empty from very beginning, then finish==True immediately
        if(self.heap_buy==[] or self.heap_sell==[]):
            finish=True

        #perform comparisons
        while(finish==False):
            if (self.heap_buy==[]):
                #this is the test case for when the sell price fails to match with any of the existing buy prices

                #reset the heap_buy list
                self.heap_buy=self.heap_buy+self.buy_holder #merge together the holder and the heap
                heapq.heapify(self.heap_buy) #sort
                self.heap_buy.reverse() #reverse the order from largest to smallest again

                #pop the current sell item onto the holder
                current=heapq.heappop(self.heap_sell)
                heapq.heappush(self.sell_holder,current)



            elif(sell=="processed" and self.heap_sell==[]):
                #this means that all possible sell combinations are already processed

                #set finish to True to end the code
                finish=True

                #no need to merge the holders back onto the heaps, because this will
                #be done during the initialization of process_orders()

            else:
                #compare the current sell order to the largest buy order

                sell=heapq.heappop(self.heap_sell)
                buy=heapq.heappop(self.heap_buy)


                if(sell[0]<=buy[0]): #this means that a transaction can be completed
                    #determine the price
                    price=sell[0]

                    #determine the quantity sold
                    sold=sell[1]-buy[1]


                    #change the string and the push actions depending on the result
                    if(sold==0):
                        #both sell order and buy order is equal
                        #nothing is pushed onto the holder stacks

                        #print the transaction
                        print(f'{buy[1]} shares sold at ${price}')

                    elif(sold>0):
                        #this means that the sell quantity is larger than the buy quantity
                        #push the remaining quantity with the same price back on to the sell heap
                        heapq.heappush(self.heap_sell,(price,sold))

                        #print the transaction
                        print(f'{buy[1]} shares sold at ${price}')


                    elif(sold<0):
```

```
                #this means that the sell quantity is smaller than the buy quantity
                #push the remaining quantity with the same price back on to the buy heap
                return_quantity=(-1)*sold
                heapq.heappush(self.heap_buy,(price,return_quantity))


                #print the transaction
                print(f'{sell[1]} shares sold at ${price}')

            #######################################
            #initialization for the next sell order
            #######################################

            #reset the heap_buy list
            self.heap_buy=self.heap_buy+self.buy_holder #merge together the holder and the heap
            heapq.heapify(self.heap_buy) #sort
            self.heap_buy.reverse() #reverse the order from largest to smallest again

            #set {sell} variable to "processed"
            sell="processed"
            #the loop now repeats until all entries in {heap_sell} are gone

        else:
        #this is the test case for when the sell_order fails to match with the current buy order
            heapq.heappush(self.buy_holder,buy) #push the current buy price onto the holder
            heapq.heappush(self.heap_sell,sell) #push the current sell price back to the heap
            sell="processed" #set sell to processed



trading = StockExchange()
trading.buy(100, 30)
trading.sell(50, 20) #Output: 50 shares sold at $20
trading.buy(50, 25)
trading.sell(100, 25) #Output: 50 shares sold at $25
```

```
50 shares sold at $20
50 shares sold at $25
```

Link to your recorded audio explanation:

https://www.veed.io/view/d6b5885f-79f5-4688-afbd-9e6c9d58744c?panel=share