# Appendix

## Question 1

import random  
import re  
from tabulate import tabulate  
  
  
class Cake:  
 def \_\_init\_\_(self, code, flavour, weight, unit\_price):  
 self.code = code  
 self.flavour = flavour  
 self.weight = weight  
 self.unit\_price = unit\_price  
  
  
class Customer:  
 cus\_id\_counter = 1 # Auto-increasing variable for generating unique customer IDs  
  
 def \_\_init\_\_(self, name, address, contact):  
 self.customer\_id = Customer.cus\_id\_counter  
 Customer.cus\_id\_counter += 1  
 self.name = name  
 self.address = address  
 self.contact\_number = contact  
  
  
class Order:  
 def \_\_init\_\_(self, customer):  
 self.order\_id = None # initialise the order\_id to None  
 self.customer = customer  
 self.cake\_items = [] # list to add multiple cakes into the order  
  
 def set\_order\_id(self, order\_id): # mutator method to set the order id (initially None)  
 self.order\_id = order\_id  
  
 def add\_cake(self, cake, weight, quantity):  
 # Function to add cake into the list in constructor  
 self.cake\_items.append((cake, weight, quantity))  
  
 def calculate\_total\_amount(self):  
 total\_amount = 0.0  
 for cake, weight, quantity in self.cake\_items:  
 cake\_price = cake.unit\_price  
 cake\_weight = weight  
 cake\_quantity = quantity  
 subtotal = cake\_price \* cake\_weight \* cake\_quantity  
 total\_amount += subtotal  
 return total\_amount  
  
  
class Node:  
 def \_\_init\_\_(self, order):  
 self.order = order  
 self.left = None  
 self.right = None  
  
  
class OrderBST:  
 def \_\_init\_\_(self):  
 self.root = None  
  
 def insert\_order(self, order):  
 if self.root is None: # if BST empty  
 self.root = Node(order) # insert the node as root  
 else: # if not empty  
 self.\_insert\_order(self.root, order)  
  
 def \_insert\_order(self, node, order):  
 # recursive method to insert data  
 if order.order\_id < node.order.order\_id: # if the value is smaller than the current node  
 if node.left is None: # the left side of the node is empty  
 node.left = Node(order) # insert the order to left subtree as a leaf  
 else: # if the node have left subtree  
 self.\_insert\_order(node.left, order) # traverse the node to the left until find the position  
 elif order.order\_id > node.order.order\_id: # if the value is bigger than the current node  
 if node.right is None:  
 node.right = Node(order)  
 else:  
 self.\_insert\_order(node.right, order)  
  
 def search\_order(self, order\_id):  
 return self.\_search\_order(self.root, order\_id)  
  
 def \_search\_order(self, node, order\_id):  
 # recursive method to search order  
 if node is None or node.order.order\_id == order\_id: # Return None if empty or don't have the order ID  
 return node.order if node else None # Return node.order if the order ID is found  
 if order\_id < node.order.order\_id: # if the order ID is smaller than the node order ID  
 return self.\_search\_order(node.left, order\_id) # move the node to left and check again  
 else: # if the order ID is larger than the node order ID  
 return self.\_search\_order(node.right, order\_id) # move the node to right and check again  
  
 def display\_all\_order\_ids(self):  
 if self.root is None: # if the BST is empty  
 print("There are no orders.")  
 return  
 self.\_display\_all\_order\_ids(self.root)  
  
 def \_display\_all\_order\_ids(self, node):  
 if node is not None: # print the order IDs that are in the BST using in-order traversal  
 self.\_display\_all\_order\_ids(node.left)  
 order = node.order  
 print(f"Order ID: {order.order\_id}\tCustomer Name: {order.customer.name}\t"  
 f"Total Amount: RM {order.calculate\_total\_amount():.2f}")  
 self.\_display\_all\_order\_ids(node.right)  
  
 def view\_orders\_details(self, current\_node, order):  
 # display all the order details for selected order ID using in-order traversal  
 if current\_node is not None: # if the current node is not empty  
 self.view\_orders\_details(current\_node.left, order)  
 # the current node traversal from the left child to search on the order id  
 if current\_node.order.order\_id == order.order\_id: # when the order id is matched, print the order details  
 print("\n----------------------------------------------------------------------------------------------")  
 print(f"Order ID: {current\_node.order.order\_id}")  
 print("--- Customer Details ---")  
 print(f"Customer ID: {current\_node.order.customer.customer\_id}")  
 print(f"Name: {current\_node.order.customer.name}")  
 print(f"Address: {current\_node.order.customer.address}")  
 print(f"Contact Number: {current\_node.order.customer.contact\_number}")  
 print("\n--- Cake Order Details ---")  
 for cake, weight, quantity in current\_node.order.cake\_items:  
 print(f"Cake Code: {cake.code}")  
 print(f"Flavour: {cake.flavour}")  
 print(f"Weight: {weight} kg")  
 print(f"Quantity: {quantity}")  
 print("")  
 print(f"Total Amount: RM {current\_node.order.calculate\_total\_amount():.2f}")  
 print("----------------------------------------------------------------------------------------------")  
 self.view\_orders\_details(current\_node.right, order)  
 # the current node traversal from the right child to search on the order id  
  
 def modify\_order(self, order\_id, new\_cake\_code, new\_flavour, new\_weight, new\_quantity, new\_unit\_price,  
 new\_customer\_name, new\_customer\_address, new\_contact):  
 # modify the details of a specific order  
 self.\_modify\_recursive(self.root, order\_id, new\_cake\_code, new\_flavour, new\_weight, new\_quantity,  
 new\_unit\_price, new\_customer\_name, new\_customer\_address, new\_contact)  
  
 def \_modify\_recursive(self, current\_node, order\_id, new\_cake\_code, new\_flavour, new\_weight, new\_quantity,  
 new\_unit\_price, new\_customer\_name, new\_customer\_address, new\_contact):  
 # private helper method  
 if current\_node.order.order\_id == order\_id: # when the order ID is found  
 total\_amount = 0.0 # assign the total amount to 0 so that can recalculate the total  
 for i, (cake, weight, quantity) in enumerate(current\_node.order.cake\_items):  
 # to update the cake lists in the order by assigning the new value  
 if cake.code == new\_cake\_code:  
 # Update cake details base on the cake code  
 cake.code = new\_cake\_code  
 cake.flavour = new\_flavour  
 cake.unit\_price = new\_unit\_price  
 current\_node.order.cake\_items[i] = (cake, new\_weight, new\_quantity)  
 # update the order cake details based on new value into the tuple list  
 subtotal = cake.unit\_price \* new\_weight \* new\_quantity  
 total\_amount += subtotal  
 current\_node.order.total\_amount = total\_amount  
 current\_node.order.customer\_name = new\_customer\_name  
 current\_node.order.customer\_address = new\_customer\_address  
 current\_node.order.customer.contact\_number = new\_contact  
  
 elif order\_id < current\_node.order.order\_id:  
 self.\_modify\_recursive(current\_node.left, order\_id, new\_cake\_code, new\_flavour, new\_weight, new\_quantity,  
 new\_unit\_price, new\_customer\_name, new\_customer\_address, new\_contact)  
 else:  
 self.\_modify\_recursive(current\_node.right, order\_id, new\_cake\_code, new\_flavour, new\_weight, new\_quantity,  
 new\_unit\_price, new\_customer\_name, new\_customer\_address, new\_contact)  
  
 def delete\_order(self, order\_id):  
 self.root = self.\_delete\_order(self.root, order\_id)  
  
 def \_delete\_order(self, node, order\_id):  
 # To find the target node  
 if node is None: # if the order ID not found  
 return node  
 if order\_id < node.order.order\_id:  
 node.left = self.\_delete\_order(node.left, order\_id)  
 elif order\_id > node.order.order\_id:  
 node.right = self.\_delete\_order(node.right, order\_id)  
 else: # Target found  
 # Node with no child or only one child  
 if node.left is None: # The node does not have left child ( 1 right child )  
 temp = node.right # temp (pointer in C++) point to the right child  
 node = None # delete the node  
 return temp # the right child become the new node  
 elif node.right is None: # The node does not have right child ( 1 left child )  
 temp = node.left  
 node = None  
 return temp  
 # The target node has two children  
 temp = self.\_min\_value\_node(node.right) # Find the successor node (smallest in the right subtree)  
 node.order = temp.order # replaces the target node's content with the order of the successor node  
 node.right = self.\_delete\_order(node.right, temp.order.order\_id) # Delete the inorder successor  
 return node  
  
 @staticmethod # does not depend on any instance-specific data and does not modify the state of the object  
 def \_min\_value\_node(node):  
 # to find the smallest value of order ID  
 current = node  
 while current.left is not None: # loop down to find the smallest value leaf  
 current = current.left  
 return current  
  
  
class CakeOrderingSystem:  
 def \_\_init\_\_(self):  
 self.bst = OrderBST()  
 self.cake\_lists = [] # List to store available cake objects  
  
 @staticmethod  
 def display\_menu():  
 print(" Hi~ o(\*￣▽￣\*)ブ Le Grande Cake Ordering System ")  
 print("1. View Cake Lists")  
 print("2. Place an Order")  
 print("3. View All Order IDs, Customer Names and Total Amount")  
 print("4. View Selected Order Details")  
 print("5. Modify an Order")  
 print("6. Delete an Order")  
 print("7. Exit")  
  
 def generate\_order\_id(self):  
 while True:  
 order\_id = random.randint(100, 10000)  
 if not self.bst.search\_order(order\_id): # so that the order ID does not duplicate  
 return order\_id  
  
 def available\_cake\_list(self):  
 # Create cake objects and add them to the cake list  
 cake1 = ["1", "Belgium Chocolate Cheesecake", 1.0, 115.00]  
 cake2 = ["2", "Burnt Cheesecake", 1.0, 95.00]  
 cake3 = ["3", "Strawberry Shortcake", 1.0, 120.00]  
 cake4 = ["4", "French Earl Grey", 1.0, 98.50]  
 cake5 = ["5", "Lemon Tart", 1.0, 100.50]  
 cake6 = ["6", "Lemon Poppy Seed", 1.0, 97.80]  
 cake7 = ["7", "Black Forest", 1.0, 96.70]  
 cake8 = ["8", "White Forest", 1.0, 96.70]  
 cake9 = ["9", "Matchamisu", 1.0, 130.00]  
 cake10 = ["10", "Tiramisu (contain alcohol)", 1.0, 135.00]  
 cake11 = ["11", "Red Velvet", 1.0, 89.00]  
 cake12 = ["12", "Blueberry Cheesecake", 1.0, 128.50]  
  
 self.cake\_lists = [cake1, cake2, cake3, cake4, cake5, cake6, cake7, cake8, cake9, cake10, cake11, cake12]  
  
 def view\_cake\_list(self):  
 # print the cake lists in table  
 # print("")  
 self.available\_cake\_list()  
 headers = ["Cake Code", "Flavour", "Weight (kg)", "Unit Price (RM/kg)"]  
 table = tabulate(self.cake\_lists, headers=headers, tablefmt="grid")  
 print(table)  
  
 def get\_cake\_info(self, cake\_code):  
 # to get the cake details for specific cake code (use in modify cake order)  
 for cake in self.cake\_lists:  
 if cake[0] == cake\_code:  
 flavour = cake[1]  
 unit\_price = cake[3]  
 return flavour, unit\_price  
 return None  
  
 def place\_order(self):  
 print("\n~~~~~ Place an Order ~~~~~")  
 print("--- Customer Details ---")  
 customer\_name = input("Enter Customer Name: ")  
 customer\_address = input("Enter Customer Address: ")  
 customer\_contact = None  
  
 # Validate customer details input must be filled  
 while not customer\_name or not customer\_address:  
 print("Customer details cannot be empty. Please try again.")  
 customer\_name = input("Enter Customer Name: ")  
 customer\_address = input("Enter Customer Address: ")  
  
 while customer\_contact is None:  
 customer\_contact = input("Contact number: ")  
  
 # Validate contact number format  
 if not re.match(r'^0\d{9,10}$', customer\_contact):  
 print("Invalid contact number. Please try again. "  
 "Contact number must start with '0' and have 10-11 digits.")  
 customer\_contact = None  
  
 # create an object called new\_customer for Customer class and pass the user input attributes to the class  
 new\_customer = Customer(customer\_name, customer\_address, customer\_contact)  
 # create new\_order object and pass it into Order class  
 new\_order = Order(new\_customer)  
  
 new\_order.set\_order\_id(self.generate\_order\_id()) # set the order id using random generated id  
 # by calling the set\_order\_id function in the Order class  
  
 print("")  
 self.view\_cake\_list() # display the cake lists  
 print("\n--- Cake Order Details ---")  
  
 while True:  
 cake\_code = input("\nEnter Cake Code: ")  
 cake = None # initialise the cake to None  
 for c in self.cake\_lists:  
 if c[0] == cake\_code: # the cake code entered is exists  
 cake = Cake(c[0], c[1], c[2], c[3]) # create an object cake and pass it into the Cake class  
 break  
  
 if cake is None: # not selecting any cake or invalid cake code  
 print("Invalid Cake Code. Please try again.")  
 continue  
  
 weight = None # initialise the weight to None  
 print("\nAvailable Weight (kg): 0.25, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0 \n")  
 while weight is None:  
 try:  
 weight = float(input("Enter Weight (in kg): "))  
 if weight not in [0.25, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0]:  
 raise ValueError  
 except ValueError:  
 print("Invalid Weight. Please enter a valid weight from the options.\n")  
 weight = None  
  
 quantity = None  
 while quantity is None:  
 try:  
 quantity = int(input("Enter Quantity: "))  
 if quantity <= 0: # validate the quantity do not get the negative value or 0  
 raise ValueError  
 except ValueError:  
 print("Invalid Quantity. Please enter a positive integer.")  
 quantity = None  
  
 new\_order.add\_cake(cake, weight, quantity) # add the cake into the list that declare in the Order class  
  
 choice = input("\nDo you want to add another cake? (Press 'y' if yes): ")  
 if choice.lower() != "y": # any input beside of y will exit the loop to add multiple cake  
 break  
  
 # Calculate and display total amount  
 total\_amount = new\_order.calculate\_total\_amount()  
 print(f"Total Amount: RM {total\_amount:.2f}")  
  
 # Insert order to BST  
 self.bst.insert\_order(new\_order)  
 print("Order Placed Successfully!\n")  
 self.bst.view\_orders\_details(self.bst.root, new\_order)  
 input("\nPress Enter to continue...")  
  
 def view\_all\_ordersID(self):  
 # display all orders in BST  
 print("\n~~~~~ All Orders ~~~~~")  
 self.bst.display\_all\_order\_ids()  
 input("\nPress Enter to continue...")  
  
 def view\_order\_details(self):  
 # display selected order in the BST  
 print("\n~~~~~ Order Details ~~~~~")  
  
 if self.bst.root is None: # if the BST is empty  
 print("There are no orders.")  
 input("\nPress Enter to continue...")  
 return  
  
 while True:  
 try:  
 order\_id = int(input("\nEnter Order ID (Press 0 to cancel): "))  
 if order\_id < 0: # if negative number  
 raise ValueError  
 if order\_id == 0:  
 return # Exit the function or method  
 except ValueError:  
 print("Invalid Order ID. Please enter a positive integer.")  
 continue  
  
 # Search the order in BST  
 order = self.bst.search\_order(order\_id)  
 if order is not None:  
 # Order found  
 self.bst.view\_orders\_details(self.bst.root, order) # print order details  
 break  
 else:  
 print("Order not found. Please try again.")  
  
 input("\nPress Enter to continue...")  
  
 def modify\_order(self):  
 # modify the value in the order  
 print("~~~~~ Modify an Order ~~~~~")  
  
 if self.bst.root is None: # if the BST is empty  
 print("There are no orders.")  
 input("\nPress Enter to continue...")  
 return  
  
 while True:  
 try:  
 order\_id = int(input("\nEnter Order ID (Press 0 to cancel): "))  
 if order\_id < 0: # if negative number  
 raise ValueError  
 if order\_id == 0:  
 return # Exit the function2  
 except ValueError:  
 print("Invalid Order ID. Please enter a positive integer.")  
 continue  
  
 # Search the order in BST  
 order = self.bst.search\_order(order\_id)  
 if order is not None: # Order found  
 self.bst.view\_orders\_details(self.bst.root, order) # print order details  
  
 # Prompt the user for the new details  
 print("\n--- Modify Customer's Details ---")  
 new\_name = input("Enter new customer name (leave blank to keep current): ")  
 new\_address = input("Enter new customer address (leave blank to keep current): ")  
 new\_contact = input("Enter new customer contact number (leave blank to keep current): ")  
  
 if new\_contact != "":  
 while not re.match(r'^0\d{9,10}$', new\_contact):  
 print("Invalid contact number. Please try again. "  
 "Contact number must start with '0' and have 10-11 digits.")  
 new\_contact = input("\nEnter new customer contact number (leave blank to keep current): ")  
 order.customer.contact\_number = new\_contact  
  
 # Update the customer details  
 if new\_name != "":  
 order.customer.name = new\_name  
 if new\_address != "":  
 order.customer.address = new\_address  
  
 # Update cake items  
 choice = input("\nDo you want to modify the cake items? (Press 'y' if yes): ")  
 if choice.lower() == "y":  
 print("\n--- Modify Cake Items ---")  
 self.view\_cake\_list() # display cake lists  
 for i, (cake, weight, quantity) in enumerate(order.cake\_items):  
 print(f"\nCake Item {i + 1}:")  
 while True:  
 new\_cake\_code = input("Enter New Cake Code (Press Enter to keep current): ")  
 if new\_cake\_code: # if the new cake code enter  
 # Get the flavour and unit price based on the cake code  
 cake\_info = self.get\_cake\_info(new\_cake\_code)  
 if cake\_info: # based on the cake info for that cake code update the cake details  
 flavour, unit\_price = cake\_info  
 cake.code = new\_cake\_code  
 cake.flavour = flavour  
 cake.unit\_price = unit\_price  
 print("\nCake details updated!")  
 break # Exit the while loop  
 else:  
 print("Invalid cake code. Please try again.\n")  
 else:  
 print("The cake is remain unchanged.")  
 break # Exit the while loop is keep the same value   
  
 print("\nAvailable Weight (kg): 0.25, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0 \n")  
  
 # update new weight  
 while True:  
 new\_weight = input("Enter New Weight (Press Enter to keep current): ")  
 if new\_weight != "":  
 try:  
 if new\_weight in ['0.25', '0.5', '1', '1.0', '1.5', '2', '2.0', '2.5', '3', '3.0']:  
 weight = float(new\_weight) # update the new weight to weight  
 break # Exit the loop if valid weight is entered  
 else:  
 raise ValueError  
 except ValueError:  
 print("Invalid Weight. Please enter a valid weight from the options.\n")  
 else:  
 print("The current weight is remain unchanged.\n")  
 break  
  
 # update new quantity  
 while True:  
 new\_quantity = input("Enter New Quantity (Press Enter to keep current): ")  
 if new\_quantity != "":  
 try:  
 new\_quantity\_int = int(new\_quantity) # covert the string entered to int  
 if new\_quantity\_int >= 1:  
 quantity = int(new\_quantity\_int) # update the new quantity to quantity  
 break # Exit the loop if valid quantity is entered  
 else:  
 raise ValueError  
 except ValueError:  
 print("Invalid Quantity. Please enter a positive integer.\n")  
 else:  
 print("The current quantity is remain unchanged.\n")  
 break  
  
 # update the cake details for the order in the list  
 order.cake\_items[i] = (cake, weight, quantity)  
  
 # Recalculate total amount  
 order.calculate\_total\_amount()  
  
 print("Order Modified Successfully!")  
 print(f"\nUpdated Order Details for Order ID {order\_id}:")  
 self.bst.view\_orders\_details(self.bst.root, order)  
 break  
 else:  
 print("Order not found. Please try again.")  
 continue  
 input("\nPress Enter to continue...")  
  
 def delete\_order(self):  
 print("~~~~~ Delete an Order ~~~~~")  
  
 if self.bst.root is None: # if the BST is empty  
 print("There are no orders.")  
 input("\nPress Enter to continue...")  
 return  
  
 while True:  
 try:  
 order\_id = int(input("Enter Order ID (Press 0 to cancel): "))  
 if order\_id < 0: # if negative number  
 raise ValueError  
 if order\_id == 0: # Exit the loop  
 return  
 except ValueError:  
 print("Invalid Order ID. Please enter a positive integer.\n")  
 continue  
  
 # Search the order in BST  
 order = self.bst.search\_order(order\_id)  
 if order is not None:  
 # Order found  
 self.bst.view\_orders\_details(self.bst.root, order) # print order details  
 confirm = input("Are you sure want to delete this order? (Press 'y' if yes): ")  
 if confirm.lower() == "y":  
 self.bst.delete\_order(order\_id)  
 print("\nOrder Deleted Successfully!")  
 break  
 else:  
 print("\nDeletion Cancelled.")  
 break  
 else:  
 print("Order not found. Please try again.")  
  
 input("\nPress Enter to continue...")  
  
 def run(self):  
 while True:  
 self.display\_menu()  
 choice = input("Enter your choice (1-7): ")  
 if choice == "1":  
 self.view\_cake\_list()  
 input("\nPress Enter to continue...")  
 elif choice == "2":  
 self.place\_order()  
 elif choice == "3":  
 self.view\_all\_ordersID()  
 elif choice == "4":  
 self.view\_order\_details()  
 elif choice == "5":  
 self.modify\_order()  
 elif choice == "6":  
 self.delete\_order()  
 elif choice == "7":  
 print("Exiting the program... ┏(＾0＾)┛ Bye~Bye~")  
 break  
 else:  
 print("Invalid choice. Please try again.\n")  
  
  
# Create and run the Cake Ordering System  
system = CakeOrderingSystem()  
system.run()

## Question 2

import random  
  
  
class HashTable:  
 def \_\_init\_\_(self, size):  
 self.\_\_size = size  
 self.\_\_chaining\_table = [[] for \_ in range(size)] # creates an empty lists for chaining  
 self.\_\_open\_addressing\_table = [None] \* size  
 self.total\_collision\_chaining = 0 # variable assign to calculate the total collision  
 self.total\_collision\_open\_addressing = 0  
  
 def \_\_hash(self, key):  
 return key % self.\_\_size # return the remainder of the value / size of hash table  
  
 def insert\_chaining(self, key): # insert the key (remainder) using chaining  
 index = self.\_\_hash(key)  
 self.\_\_chaining\_table[index].append(key)  
  
 def insert\_open\_addressing(self, key): # insert the key (remainder) using open addressing  
 index = self.\_\_hash(key)  
 while self.\_\_open\_addressing\_table[index] is not None: # if collision occurs  
 self.total\_collision\_open\_addressing += 1 # calculate the total collision of open addressing  
 index = (index + 1) % self.\_\_size # keep increasing the index by 1 to find the available slot to insert  
 self.\_\_open\_addressing\_table[index] = key # insert the key into after found the available slot  
  
 def calculate\_total\_collision\_chaining(self): # calculate the total collisions in chaining  
 for n in self.\_\_chaining\_table:  
 if len(n) > 1: # if there has element(s) in the hash table  
 self.total\_collision\_chaining += len(n) - 1 # increase the counter with the length of slot and  
 # subtract 1 to exclude the first key that was inserted without collision  
 return self.total\_collision\_chaining  
  
  
def run\_program():  
 number\_items = 5000  
 table\_size = 6001  
 num\_execute = 10 # Number of execute times  
  
 chaining\_min\_collisions = float('inf') # positive infinity value  
 chaining\_max\_collisions = float('-inf') # negative infinity value  
 chaining\_avg\_collisions = 0 # to calculate the total average of chaining for 10 execution  
  
 open\_addressing\_min\_collisions = float('inf')  
 open\_addressing\_max\_collisions = float('-inf')  
 open\_addressing\_avg\_collisions = 0 # to calculate the total average of open addressing for 10 execution  
  
 for \_ in range(num\_execute):  
 items = [random.randint(0, 100000) for \_ in range(number\_items)] # autogenerate 5000 random numbers  
  
 chaining\_table = HashTable(table\_size)  
 open\_addressing\_table = HashTable(table\_size)  
  
 for item in items: # insert the numbers into hash table  
 chaining\_table.insert\_chaining(item)  
 open\_addressing\_table.insert\_open\_addressing(item)  
  
 # calculate the total collision of every execution for collision and open addressing  
 total\_collision\_chaining = chaining\_table.calculate\_total\_collision\_chaining()  
 print(f"Total collision Chaining: {total\_collision\_chaining}")  
 print(f"Total collision Open Addressing: {open\_addressing\_table.total\_collision\_open\_addressing}\n")  
  
 # Update minimum and maximum collisions for chaining  
 chaining\_min\_collisions = min(chaining\_min\_collisions, total\_collision\_chaining)  
 chaining\_max\_collisions = max(chaining\_max\_collisions, total\_collision\_chaining)  
  
 # Update minimum and maximum collisions for open addressing  
 open\_addressing\_min\_collisions = min(open\_addressing\_min\_collisions, open\_addressing\_table.total\_collision\_open\_addressing)  
 open\_addressing\_max\_collisions = max(open\_addressing\_max\_collisions, open\_addressing\_table.total\_collision\_open\_addressing)  
  
 # Add the total collisions to calculate the average after every execution  
 chaining\_avg\_collisions += total\_collision\_chaining  
 open\_addressing\_avg\_collisions += open\_addressing\_table.total\_collision\_open\_addressing  
  
 # Clean up hash tables after every execution  
 chaining\_table.chaining\_table = [[] for \_ in range(table\_size)]  
 open\_addressing\_table.open\_addressing\_table = [None] \* table\_size  
  
 # Calculate average collisions  
 chaining\_avg\_collisions /= num\_execute # average = total average / 10  
 open\_addressing\_avg\_collisions /= num\_execute  
  
 # Print results in tabular format  
 print("-------------------------------------------------------------------------------------")  
 print("|\t\t\t\t\t\t\t\t\tAverage cost\t\t\t\t\t\t\t\t\t|")  
 print("-------------------------------------------------------------------------------------")  
 print("|\t\t\t\tChaining\t\t\t\t|\t\t\t\tOpen Addressing\t\t\t\t|")  
 print("-------------------------------------------------------------------------------------")  
 print("|\tMin\t\t|\tMax\t\t|\t Avg\t\t|\t\tMin\t\t|\t Max\t | \tAvg\t\t|")  
 print("-------------------------------------------------------------------------------------")  
 print(f"|\t{chaining\_min\_collisions}\t|\t{chaining\_max\_collisions}\t|\t{chaining\_avg\_collisions:.2f}\t\t|\t\t"  
 f"{open\_addressing\_min\_collisions}\t|\t{open\_addressing\_max\_collisions}\t | {open\_addressing\_avg\_collisions:.2f}\t|")  
 print("-------------------------------------------------------------------------------------")  
  
  
run\_program()

## Question 3

class WeightedGraph:  
 # Constructor  
 def \_\_init\_\_(self, edges, n):  
 # allocate memory for the adjacency list  
 self.adjacencyList = [[] for \_ in range(n)]  
  
 # add edges to the directed graph  
 for (src, destination, weight) in edges:  
 # allocate node in adjacency list from src to destination with weight  
 self.adjacencyList[src].append((destination, weight))  
  
 # Function to list all adjacent Vertices  
 def listAdjacentVertex(self, vertex):  
 return [adj\_vertex for (adj\_vertex, \_) in self.adjacencyList[vertex]]  
  
 # Function to calculate the sum of all the weight for all the adjacent Vertices of a graph  
 def sumHighestAdjacentVertex(self, vertex):  
 adj\_list = self.listAdjacentVertex(vertex)  
 if not adj\_list:  
 return 0  
 return sum([weight for (\_, weight) in self.adjacencyList[vertex]])  
  
  
# Function to print adjacency list representation of a graph  
def printWeightedGraph(graph):  
 for src in range(len(graph.adjacencyList)):  
 # print current vertex and all its neighboring vertices with weights  
 for (destination, weight) in graph.adjacencyList[src]:  
 print(f'({src} —({weight})—> {destination}) ', end='')  
 print()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Edges and weights in a directed weighted graph  
 edges1 = [(0, 1, 31), (0, 6, 13),  
 (1, 2, 11), (1, 3, 9), (1, 4, 10),  
 (2, 0, 8),  
 (3, 4, 5), (3, 5, 14),  
 (4, 3, 12),  
 (5, 2, 7), (5, 3, 3)]  
 edges2 = [(0, 1, 2), (0, 2, 4), (0, 3, 6),  
 (1, 3, 8), (1, 4, 4),  
 (2, 0, 6), (2, 4, 2),  
 (3, 4, 5),  
 (4, 5, 10)]  
  
 # No. of vertices  
 n = 6  
  
 # construct a graph from a given list of edges  
 graph1 = WeightedGraph(edges1, n)  
 graph2 = WeightedGraph(edges2, n)  
  
 # print adjacency list representation of the graph  
 print("Graph 1")  
 printWeightedGraph(graph1)  
 for vertex in range(n):  
 print("\nAdjacent vertices of vertex", vertex, ":", graph1.listAdjacentVertex(vertex))  
 print("Sum of the weights of adjacent vertices of vertex", vertex, ":", graph1.sumHighestAdjacentVertex(vertex))  
  
 print("\n\nGraph 2")  
 printWeightedGraph(graph2)  
 for vertex in range(n):  
 print("\nAdjacent vertices of vertex", vertex, ":", graph2.listAdjacentVertex(vertex))  
 print("Sum of the weights of adjacent vertices of vertex", vertex, ":", graph2.sumHighestAdjacentVertex(vertex))

## Question 4

import threading  
  
  
# calculate the monthly repayment for the customer  
def calculate\_monthly\_repayment(customer, amount\_to\_loan, interest\_rate, loan\_duration):  
 interest\_rate = interest\_rate / 100 # Converting interest rate from percentage to decimal  
 loan\_period\_months = loan\_duration \* 12  
  
 total\_interest = amount\_to\_loan \* interest\_rate \* loan\_duration  
 total\_amount\_repayable = amount\_to\_loan + total\_interest  
 monthly\_repayment = total\_amount\_repayable / loan\_period\_months  
  
 # Displaying the monthly repayment  
 print(f"Calculate the monthly repayment for {customer}.")  
 print(f"The monthly repayment for a loan of RM{amount\_to\_loan:.2f} with {interest\_rate \* 100:.2f}% interest rate "  
 f"for {loan\_duration} years is RM{monthly\_repayment:.2f}.\n")  
  
  
# Main program  
def main():  
 # Input values for three customers  
 customers = []  
  
 # Accept user input for three customers  
 for i in range(3):  
 customer = {}  
 print(f"Customer {i + 1}:")  
 customer["customer"] = input("Please enter customer name: ")  
  
 while True:  
 try:  
 customer["amount\_to\_loan"] = float(input("Please enter the amount to loan: "))  
 if customer["amount\_to\_loan"] > 0:  
 break  
 else:  
 print("Invalid input. Please enter a loan value that greater than 0.")  
 except ValueError:  
 print("Invalid input. Please enter a valid number.")  
  
 while True:  
 try:  
 customer["interest\_rate"] = float(input("Please enter interest rate: "))  
 if customer["interest\_rate"] > 0:  
 break  
 else:  
 print("Invalid input. Please enter the interest rate that greater than 0.")  
 except ValueError:  
 print("Invalid input. Please enter a valid number.")  
  
 while True:  
 try:  
 customer["loan\_duration"] = int(input("Please enter loan duration (in years): "))  
 if customer["loan\_duration"] > 0:  
 break  
 else:  
 print("Invalid input. Please enter a loan period that greater than 0.")  
 except ValueError:  
 print("Invalid input. Please enter a valid integer.")  
  
 customers.append(customer) # insert the 3 customer into list  
 print("")  
  
 threads = [] # a list to save all the threads  
 for customer in customers:  
 # Create a thread for each customer and start it  
 thread = threading.Thread(target=calculate\_monthly\_repayment,  
 args=(customer["customer"], customer["amount\_to\_loan"], customer["interest\_rate"],  
 customer["loan\_duration"]))  
 thread.start() # to start the execution of the thread  
 # initially the target function in a separate thread, start() allows the threads to run concurrently  
 threads.append(thread) # insert the 3 customer thread into list  
  
 # Wait for all threads to finish then only end the program  
 for thread in threads:  
 thread.join()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()