

Section A - To be completed by the student

Student Details:					
	No.	1	2	3	4
	Full Name	Neo Jun Khai	Someshwar Rao	Lee Woei Liang	Wong Weng Kean
	CU ID Number	15096795	15096810	15387084	15387110
	Peer Assessment Rating (To be Filled by the lecturer)				
	Final Score (To be Filled by the lecturer)				
Semester: APRIL 2024					
Lecturer: Khor Jia Yun					
Module Code and Title: 5008CEM Programming for Developers					
Assignment No. / Title: Group Report			25% of Module Mark		
Hand out date: 12th April 2024 (Friday of Week 2)			Due date: 1st July 2024 (Monday of Week 14)		

Penalties: No late work will be accepted. If you are unable to submit coursework on time due to extenuating circumstances you may be eligible for an extension. Please consult the lecturer.

Declaration:

*I/we the undersigned **confirm** that I/we have **read and agree to abide** by the University **regulations on plagiarism and cheating** and Faculty coursework policies and procedures. I/we confirm that this piece of **work is my/our own**. I/we consent to appropriate storage of our work for plagiarism checking.*

Khai

Someshwar Rao

Signature(s): _____

(Student 1 Signature)

(Student 2 Signature)



KEAN

(Student 3 Signature)

(Student 4 Signature)

Section B - To be completed by the module leader

Intended learning outcomes (LO) assessed by this work:

LO2: Design and implement algorithms and data structures for novel problems.

LO3: Understand the intractability of certain problems and implement approaches to estimate the solution to intractable problems.

LO5. Design and implement a basic concurrent application.

Marking scheme	Max	Mark
<i>Refer to pages 3 & 4 of this document.</i>		
Total		
Lecturer's Feedback		

Internal Moderator's Feedback

Table of Contents

Question 1 - Neoh Jun Khai	6
Code Implementation & Program Outcome	6
Weaknesses	25
Reflection	26
Question 2 - Someshwar Rao:	26
Code Implementation & Program Outcome	27
Weaknesses	38
Reflection	39
Question 3 - Lee Woei Liang:	41
Code Implementation	41
Program Outcome	47
Weaknesses	50
Reflection	51
Question 4 - Wong Weng Kean:	52
Code Implementation & Program Outcome	52
Weaknesses	58
Reflection	59
Appendix	61
Question 1 Code Solution	61
'smartphone.py' file	61
'bst.py' file	63
'main.py' file	66
Question 2 Code Solution	74
2(a)	74
2(b)	77
'main1.py' file	77
'main2.py' file	84
Question 3 Code Solution	92
Question 4 Code Solution	96

Question 1 - Neoh Jun Khai

Code Implementation & Program Outcome

In question 1 of the assignment, we are required to design and develop a Switch Store Stock Management System to manage only one product of our choice. The Switch Store Stock Management System I have designed is to efficiently manage the inventory of smartphones. The implementation leverages a Binary Search Tree (BST) data structure to facilitate quick and organized access to the inventory, ensuring optimal performance even as the dataset grows. This report details the design and functionality of the system, highlighting the use of object-oriented programming (OOP) principles, data validation, and error handling. This system is **menu driven**, giving the user various choices of operation that allow the user to:

- 1. Add product**
- 2. View all products**
- 3. Search for a particular product based on the brand**
- 4. Modify the details of a product**

Moreover, I have **added the following feature**:

- 1. Search for a particular product based on the product code**

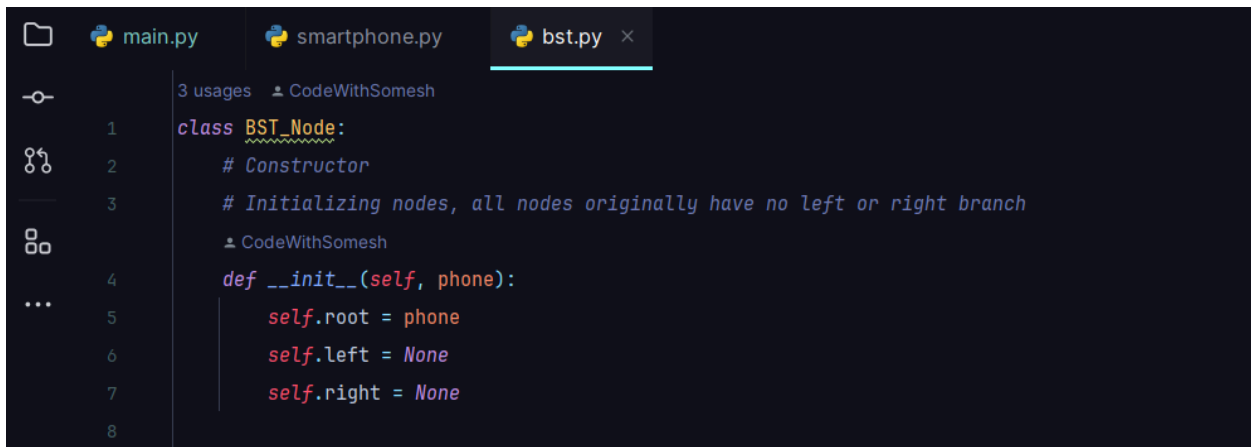


```
main.py | smartphone.py | bst.py
CodeWithSomesh
1 class Smartphone:
2     # Constructor
3     # Initializing phones, all nodes originally have all these attributes
4     CodeWithSomesh
5     def __init__(self, productCode, brand, model, sellingPrice, colorArray, quantityOnHandArray, serialNumber):
6         self.productCode = productCode
7         self.brand = brand
8         self.model = model
9         self.sellingPrice = sellingPrice
10        self.colorArray = colorArray
11        self.quantityOnHandArray = quantityOnHandArray
12        self.serialNumber = serialNumber
13
14    # The __str__ method is called to get a user-friendly string representation of the object
15    # Particularly useful for debugging and logging
16    CodeWithSomesh
17    def __str__(self):
18        allColors = ', '.join(self.colorArray)
19        allQuantity = ', '.join(self.quantityOnHandArray)
20
21        return (f"Product Code: {self.productCode}, Brand: {self.brand}, Model: {self.model}, "
22                f"Selling Price: RM{self.sellingPrice:.2f}, Serial Number: {self.serialNumber}"
23                f"\nAll Colors: {allColors} \nAll Quantity: {allQuantity}")
24
25    # Updating details of the phone when user select Modify Choice in Menu
26    CodeWithSomesh
27    def updateDetails(self, newPhoneDetails):
28        self.productCode = newPhoneDetails.productCode
29        self.brand = newPhoneDetails.brand
30        self.model = newPhoneDetails.model
31        self.sellingPrice = newPhoneDetails.sellingPrice
32        self.colorArray = newPhoneDetails.colorArray
33        self.quantityOnHandArray = newPhoneDetails.quantityOnHandArray
34        self.serialNumber = newPhoneDetails.serialNumber
35        return self
```

Figure 1 & 2: Smartphone Class in 'smartphone.py' file

The Smartphone class models the essential attributes of a smartphone, encapsulating details such as product code, brand, model, selling price, available colors, quantities on hand, and serial number. This class includes methods to initialize a smartphone object, **provide a string representation for easy display**, and **update the details of an existing smartphone**.

- **Constructor:** Initializes a new smartphone with the provided attributes.
- **“__str__” Method:** Returns a formatted string representation of the smartphone, useful for displaying product details.
- **“updateDetails” Method:** Updates the attributes of an existing smartphone with new values provided by the user.



```
3 usages  CodeWithSomesesh
1  class BST_Node:
2      # Constructor
3      # Initializing nodes, all nodes originally have no left or right branch
4      CodeWithSomesesh
5      def __init__(self, phone):
6          self.root = phone
7          self.left = None
8          self.right = None
```



```
CodeWithSomesesh
class BST:
    # Constructor
    # Initializing BST Tree, BST trees originally are always empty with no Root Node
    CodeWithSomesesh
    def __init__(self):
        self.root = None

CodeWithSomesesh
def addChild(self, phone, node):
    # Checking to see whether there is already a Root Node
    if self.root is None:
        self.root = BST_Node(phone) # If no Root Node then make the first value Root Node
    else:
        # If got Root Node, but current product code is BIGGER than Root Node
        if phone.productCode > node.root.productCode:
            # And if there is no RIGHT CHILD for this node, then make the current phone the RIGHT CHILD
            if node.right is None:
                node.right = BST_Node(phone)
            # If not run this same method recursively until it becomes a RIGHT CHILD
            else:
                self.addChild(phone, node.right)
        # If got Root Node, but current product code is SMALLER than Root Node
        elif phone.productCode < node.root.productCode:
            # And if there is no LEFT CHILD for this node, then make the current phone the LEFT CHILD
            if node.left is None:
                node.left = BST_Node(phone)
            # If not run this same method recursively until it becomes a LEFT CHILD
            else:
```

```
15     def addChild(self, phone, node):
16         # And if there is no LEFT CHILD for this node, then make the current phone the LEFT CHILD
17
18         if node.left is None:
19             node.left = BST_Node(phone)
20             # If not run this same method recursively until it becomes a LEFT CHILD
21         else:
22             self.addChild(phone, node.left)
23         else:
24             return False
25
26
27
28
29     # Using LVR to return an array of elements (phones)
30     10 usages (8 dynamic)  CodeWithSomesesh
31     def inOrderTraversal(self, node):
32         # Initialize empty array
33         elements = []
34
35         if node is None:
36             return None
37
38         # Visit Left node first, then recursively find for the furthest left leaf node,
39         # then add in the array
40         if node.left:
41             elements += self.inOrderTraversal(node.left)
42
43         # After adding the left node value of the node, then add the root node value
44         elements.append(node.root)
45
46         # Lastly add the right node recursively
47         if node.right:
48             elements += self.inOrderTraversal(node.right)
```

```
CodeWithSomesesh
def search(self, productCode, node):

    if node is None:
        return None

    if productCode == node.root.productCode:
        return node.root

    # If the current phone productCode is BIGGER than node's,
    # then run this same method recursively until the phone with given product code is found
    elif productCode > node.root.productCode:
        return self.search(productCode, node.right)

    # If the current phone productCode is SMALLER than node's,
    # then run this same method recursively until the phone with given product code is found
    elif productCode < node.root.productCode:
        return self.search(productCode, node.left)

    else:
        return None # Add Print Statement
```

```
CodeWithSomesesh
def modify(self, productCode, newPhoneDetails):
    existingPhone = self.search(productCode, self.root)
    # print(existingPhone)
    modifiedPhone = existingPhone.updateDetails(newPhoneDetails)
    # print(modifiedPhone)
    return modifiedPhone
```

Figure 3 - 7: BST and BST_Node Classes in 'bst.py' file

The Binary Search Tree (BST) is implemented using two classes: BST_Node and BST. The BST_Node class represents a single node in the tree, containing a smartphone object and pointers to its left and right children. The BST class manages the overall structure of the tree and includes methods for **adding**, **searching**, **modifying**, and **traversing nodes**.

- **BST_Node Class:**
 - **Constructor:** Initializes a node with a smartphone object and sets left and right children to None.

- **BST Class:**
 - **Constructor:** Initializes an empty tree.
 - **“addChild” Method:** Adds a new smartphone to the tree, maintaining the BST property (left child nodes are smaller, right child nodes are larger).
 - **“inOrderTraversal” Method:** Performs an in-order traversal of the tree, returning a list of smartphones in ascending order by product code.
 - **“search” Method:** Searches for a smartphone by product code, returning the corresponding node if found.
 - **“modify” Method:** Updates the details of an existing smartphone in the tree.

These classes work together to provide a robust and efficient stock management system, allowing users to perform various operations on the smartphone inventory with ease.

By utilizing these classes and methods, the Switch Store Stock Management System ensures an organized and efficient way to manage smartphone inventory, providing users with a powerful tool to handle their stock effectively.

```
CodeWithSomesesh
211 def menu(bst):
212     while True:
213         print("\nWelcome To Switch Store Stock Management System")
214         print("\nAvailable Operations: ")
215         print("1. Create a new product")
216         print("2. View all products")
217         print("3. Search for a phone by product code")
218         print("4. Search for phones by phone brand")
219         print("5. Modify phone details")
220         print("6. Exit")
221         choice = input("\nEnter your choice: ")
222         if choice == '1':
223             os.system('cls')
224             print("Create a new product: ")
225             addPhone(bst)
226             print()
227             os.system('pause')
228             os.system('cls')
229         elif choice == '2':
230             os.system('cls')
231             print("View all products: ")
232             viewAllPhones(bst)
233             print()
234             os.system('pause')
235             os.system('cls')
236         elif choice == '3':
237             os.system('cls')
238             print("Search for a phone by product code: ")
239             searchPhoneByProductCode(bst)
```

Figure 8:

The function that displays the Menu as well as the functions that are carried out when each operation is selected

```
Welcome To Switch Store Stock Management System

Available Operations:
1. Create a new product
2. View all products
3. Search for a phone by product code
4. Search for phones by phone brand
5. Modify phone details
6. Exit

Enter your choice: █
```

Figure 9: Menu that is displayed for end user

```
2 usages  CodeWithSomesesh
5 def getFloatInput(prompt, existingPhone=None):
6     while True:
7         userInput = input(prompt) or existingPhone.sellingPrice
8         try:
9             # Try to convert the input to a float
10            num = float(userInput)
11            return num
12
13        except ValueError:
14            # If conversion fails, it's not a valid number
15            print("Invalid input. Please enter a valid number.")
16
```

```
4 usages  🔍 CodeWithSomesesh
def getIntInput(prompt):
    while True:
        userInput = input(prompt)
        if userInput.isdigit():
            return int(userInput)
        else:
            print("Invalid input. Please enter a valid integer number.")

1 usage  🔍 CodeWithSomesesh
def getYesOrNoInput(prompt):
    while True:
        userInput = input(prompt).upper()
        if userInput == 'Y' or userInput == 'YES':
            return True
        elif userInput == 'N' or userInput == 'NO':
            return False
        else:
            print("Invalid input. Please enter 'Yes' or 'No' only.")
```

Figure 10 & 11: Validating functions in “main.py” file

```
Enter Phone Selling Price: RMa
Invalid input. Please enter a valid number.
Enter Phone Selling Price: RM_
```

```
Enter Number of Colorways Available: s
Invalid input. Please enter a valid integer number.
Enter Number of Colorways Available:
```

```
Do you want to modify the Phone Color and Quantity? (Yes/No): 123
Invalid input. Please enter 'Yes' or 'No' only.
Do you want to modify the Phone Color and Quantity? (Yes/No): _
```

Figure 12 - 14: Program output for the validating functions

The validations in Figure 10 and 11 ensures the users can reenter the inputs if they have mistakenly entered anything invalid. The 'getFloatInput' function is to ensure users only enter any number input with decimal, if user enters anything else aside numbers, it will display the same message as shown in Figure 12, then it reprompts the user to enter valid number input. When users are requested any integer input and they entered something invalid like show in Figure 13, there will be an error message displayed and the system reprompts the user for valid data. Whereas, if an user is required to enter 'Yes' or 'No' as input and if they entered any number, there will be an error message as well as a reprompt for the user, just like shown in Figure 14.


```
1 usage  ± CodeWithSomesesh
def addPhone(bst):
    colorArray = []
    quantityArray = []

    productCode = (input("\nEnter Product Code: ")).upper()
    allPhones = bst.inOrderTraversal(bst.root)
    if allPhones is not None:
        for phones in allPhones:
            if productCode == phones.productCode:
                print("\nThere is already a phone under this Product Code.")
                print("Select 1 in the Menu to add a new phone with a different Product Code.")
                print("Select 5 in the Menu to modify the details of the phone with this Product Code.")
                return

    serialNumber = input("Enter Serial Number: ")
    brand = input("Enter Phone Brand: ")
    model = input("Enter Phone Model: ")
    sellingPrice = getFloatInput("Enter Phone Selling Price: RM")

    numberOfColors = getIntInput("Enter Number of Colorways Available: ")
    for i in range(numberOfColors):
        color = input(f"Enter Phone Color {i+1}: ")
        quantityOnHand = getIntInput(f"Enter Quantity of Phone Color {i+1}: ")
        quantityOnHand = str(quantityOnHand)
        colorArray.append(color)
        quantityArray.append(quantityOnHand)
```

Figure 15: Function that carries out Add Product feature

```
Create a new product:

Enter Product Code: at15
Enter Serial Number: abc123
Enter Phone Brand: Apple
Enter Phone Model: iPhone15
Enter Phone Selling Price: RM4599
Enter Number of Colorways Available: 3
Enter Phone Color 1: Blue
Enter Quantity of Phone Color 1: 10
Enter Phone Color 2: Pink
Enter Quantity of Phone Color 2: 23
Enter Phone Color 3: Yellow
Enter Quantity of Phone Color 3: 88

Product added successfully.

New Product Details:
Product Code: AT15, Brand: Apple, Model: iPhone15, Selling Price: RM4599.00, Serial Number: abc123
All Colors: Blue, Pink, Yellow
All Quantity: 10, 23, 88

Press any key to continue . . .
```

Figure 16: Add Product feature that is displayed

```
Create a new product:

Enter Product Code: at15

There is already a phone under this Product Code.
Select 1 in the Menu to add a new phone with a different Product Code.
Select 5 in the Menu to modify the details of the phone with this Product Code.

Press any key to continue . . .
```

Figure 17:

Error message is displayed when user tries to add a new product with an existing product code

As we can see in Figure 16, upon users successfully entering all valid inputs, the product will be successfully added in the system. On the other hand, as shown in Figure 17, when users enters a existing phone's product code while adding a new product in the system, a proper error message is displayed, the system also guides the user on how to move forward, in this case, they could select 1 in the Menu to add a new phone with a different Product Code or they could select 5 in the Menu to modify the details of the phone with the existing Product Code.

Moreover, I made sure users can enter all the colorways of a smartphone and the quantity of each colorway in one go when adding a product in the database. By adding this feature, users does not have to to always add a new product when there is a new colorway of a smartphone released, one by one, they can choose to add the product with all the colorways in one go or choose to modify the product in Option 5 of the Meu and add all the new colorways there.

```
1 usage  ▲ CodeWithSomesesh *
def viewAllPhones(bst):
    num = 1
    allPhones = bst.inOrderTraversal(bst.root) # Displaying starting from the smallest Product Code
    if allPhones is None:
        print("\nThere aren't any products added in the system")
    else:
        for phones in allPhones:
            print(f"\n***** Product {num} ***** ")
            print(f"{phones}")
            num += 1
```

Figure 18: Function that carries out View All Phones feature

```
View all products:
***** Product 1 *****
Product Code: AT15, Brand: Apple, Model: iPhone15, Selling Price: RM4599.00, Serial Number: abc123
All Colors: Blue, Pink, Yellow
All Quantity: 10, 20, 33
***** Product 2 *****
Product Code: AT21, Brand: Samsung, Model: S24, Selling Price: RM2399.00, Serial Number: bcd234
All Colors: Titanium, Peach
All Quantity: 22, 44
Press any key to continue . . .
```

Figure 19: View All Phones feature that is displayed

When users selects Number 2 in the Menu, they are able to view all the products that are added into the system.

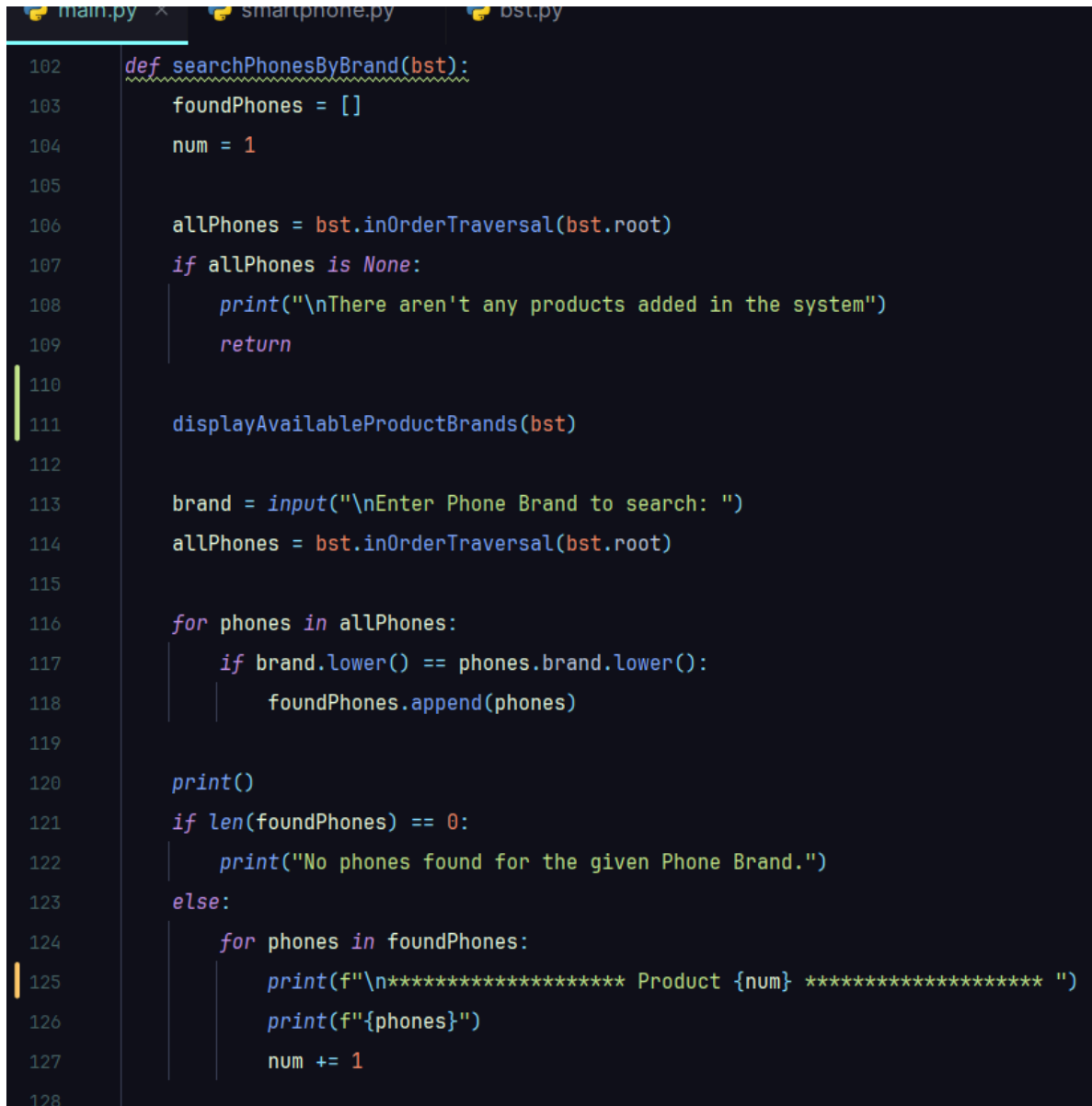
```
def searchPhoneByProductCode(bst):  
    allPhones = bst.inOrderTraversal(bst.root)  
    if allPhones is None:  
        print("\nThere aren't any products added in the system")  
        return  
  
    displayAvailableProductCodes(bst)  
  
    productCode = (input("\nEnter Product Code to search: ")).upper()  
    foundPhone = bst.search(productCode, bst.root)  
    print()  
  
    if foundPhone:  
        print(foundPhone)  
    else:  
        print("No phone found for the given Product Code.")
```

Figure 20: Function that carries out Search Phone By Product Code feature

```
Search for a phone by product code:  
***** Available Product Codes In The System *****  
1. AT15  
2. AT21  
*****  
  
Enter Product Code to search: at15  
  
Product Code: AT15, Brand: Apple, Model: iPhone15, Selling Price: RM4599.00, Serial Number: abc123  
All Colors: Blue, Pink, Yellow  
All Quantity: 10, 20, 33  
  
Press any key to continue . . .
```

Figure 21: Search Phone By Product Code feature that is displayed

The Search Phone By Product Code feature that we can see in Figure 21 is the feature I added additionally. Users are able to find for their smartphone details by just entering the product code of the smartphone. They can see what are all the Available Product Codes In The System, then select their smartphone's product code to search for it.

The image shows a code editor with three tabs: main.py, smartphone.py, and bst.py. The bst.py tab is active, displaying a Python function named searchPhonesByBrand. The function takes a bst parameter and performs a search for a phone by brand. It initializes foundPhones as an empty list and num as 1. It then calls bst.inOrderTraversal(bst.root) to get allPhones. If allPhones is None, it prints a message and returns. Otherwise, it calls displayAvailableProductBrands(bst). It prompts the user to enter a phone brand and then iterates through allPhones to find matches. If no matches are found, it prints a message. If matches are found, it prints the product code and the phone details for each match, incrementing num for each result.

```
102 def searchPhonesByBrand(bst):
103     foundPhones = []
104     num = 1
105
106     allPhones = bst.inOrderTraversal(bst.root)
107     if allPhones is None:
108         print("\nThere aren't any products added in the system")
109         return
110
111     displayAvailableProductBrands(bst)
112
113     brand = input("\nEnter Phone Brand to search: ")
114     allPhones = bst.inOrderTraversal(bst.root)
115
116     for phones in allPhones:
117         if brand.lower() == phones.brand.lower():
118             foundPhones.append(phones)
119
120     print()
121     if len(foundPhones) == 0:
122         print("No phones found for the given Phone Brand.")
123     else:
124         for phones in foundPhones:
125             print(f"\n***** Product {num} ***** ")
126             print(f"{phones}")
127             num += 1
128
```

Figure 22: Function that carries out Search Phone By Phone Brand feature

```
Search for phones by phone brand:
***** Available Phone Brands In The System *****
1. Apple
2. Samsung
*****

Enter Phone Brand to search: apple

***** Product 1 *****
Product Code: AT15, Brand: Apple, Model: iPhone15, Selling Price: RM4599.00, Serial Number: abc123
All Colors: Blue, Pink, Yellow
All Quantity: 10, 20, 33

Press any key to continue . . .
```

Figure 23: Search Phone By Phone Brand feature that is displayed

The Search Phone By Phone Brand feature that we can see in Figure 23 is very similar to the Search Phone By Product Code feature that we can see in Figure 21. Here, users are also able to find for their smartphone details by just entering the phone's brand instead. They can see what are all the Available Phone Brands In The System, then select their smartphone's brand to search for it.

```
def modifyPhoneDetails(bst):  
    colorArray = []  
    quantityArray = []  
  
    allPhones = bst.inOrderTraversal(bst.root)  
    if allPhones is None:  
        print("\nThere aren't any products added in the system")  
        return  
  
    displayAvailableProductCodes(bst)  
  
    productCode = (input("\nEnter Product Code to modify: ")).upper()  
    existingPhone = bst.search(productCode, bst.root)  
    print()  
  
    if not existingPhone:  
        print("Product not found.")  
        return  
  
    print("Current phone details:")  
    print(existingPhone)  
    print()  
  
    print("Enter new details (Leave blank and Press Enter to keep existing value):")  
    serialNumber = input(f"Enter Serial Number ({existingPhone.serialNumber}): ") or existingPhone.serialNumber  
    brand = input(f"Enter Brand ({existingPhone.brand}): ") or existingPhone.brand  
    model = input(f"Enter Model ({existingPhone.model}): ") or existingPhone.model  
    sellingPrice = getFloatInput( prompt: f"Enter Selling Price (RM{existingPhone.sellingPrice:.2f}): RM", existingPhone)  
    # sellingPrice = float(sellingPrice) if sellingPrice else existingPhone.sellingPrice # Need make changes
```

Figure 24: Function that carries out Modify Product Details feature

```
Modify phone details:

***** Available Product Codes In The System *****
1. AT15
2. AT21
*****

Enter Product Code to modify: at15

Current phone details:
Product Code: AT15, Brand: Apple, Model: iPhone15, Selling Price: RM4599.00, Serial Number: abc123
All Colors: Blue, Pink, Yellow
All Quantity: 10, 20, 33

Enter new details (Leave blank and Press Enter to keep existing value):
Enter Serial Number (abc123):
Enter Brand (Apple):
Enter Model (iPhone15):
Enter Selling Price (RM4599.00): RM6599
Do you want to modify the Phone Color and Quantity? (Yes/No): n

Product details updated successfully.

Newly modified phone details:
Product Code: AT15, Brand: Apple, Model: iPhone15, Selling Price: RM6599.00, Serial Number: abc123
All Colors: Blue, Pink, Yellow
All Quantity: 10, 20, 33

Press any key to continue
```

Figure 25: Modify Product Details feature that is displayed

Users are able to modify the smartphone details by selecting the 5th option in the Menu. Users are able to see the available product codes in the system before entering the product code of a smartphone in order to modify them. First the existing details of the product will be displayed then the system will ask the users for new details for their product. If the user wish to keep their existing product details, then they just have to press Enter. Once all valid details are entered, the product details will be modified successfully and the new product details will be displayed as well, just like show in Figure 25.

Weaknesses

As the number of smartphones in the system increases, the performance of the Binary Search Tree (BST) might degrade if the tree becomes unbalanced. This could lead to inefficient operations with time complexity approaching $O(n)$ in the worst case.

Moreover, the system currently handles only smartphones. Extending the system to manage other types of electronic devices (e.g., tablets, smart watches) or adding features like sales tracking and reporting could enhance its functionality.

Reflection

From this project, I have gained a deeper understanding of object-oriented programming by implementing classes and methods to represent and manipulate product management in a binary search tree (BST) structure. Creating the `Product`, `TreeNode`, and `BST` classes improved my skills in designing and implementing classes in Python. The `Product` class encapsulates product attributes, while the `BST` class manages the overall tree structure, including nodes, insertions, and searches. This modular design makes the code more readable, maintainable, and scalable.

Implementing the `insert` function taught me how to manage nodes in a BST, including adding and checking for existing nodes efficiently and maintaining the tree structure. I learned the importance of maintaining data integrity and avoiding redundant entries by carefully checking for duplicate product codes before adding new products.

Through this project, I recognized the importance of robust error handling and input validation. Although the current implementation has minimal error handling, I understand the need for comprehensive checks to ensure the program can gracefully handle unexpected inputs and conditions. For example, ensuring that numeric fields like selling price and quantity are correctly validated to prevent crashes or incorrect data entries. The process of developing and refining the code highlighted the importance of iterative development. Some of the functions and methods were improved through multiple iterations, which involved identifying weaknesses, testing different approaches, and refining the implementation to achieve better performance and reliability. This iterative approach helped me understand the value of continuous improvement and testing in software development. By implementing a menu-driven interface to create products, view all products, search for products by brand, and modify product details, I learned how to design interactive programs that respond to user inputs. This experience has been invaluable in understanding how to create user-friendly interfaces and handle user interactions effectively. Overall, this project has been a significant learning experience in object-oriented programming, data structures, and user interface design, providing me with practical skills and insights that will be beneficial in future software development endeavors.

Question 2 - Someshwar Rao:

Code Implementation & Program Outcome

In section (a) of Question 2, we are required to create a hash function that generates a hash code from a person's Malaysian IC Number (12 digits without the dash). The hash function will use an appropriate folding technique and generate the hash code based on the IC number.

```
31  # My Hash Function with my Custom Folding Technique
    CodeWithSomesh
32  def hashFunction(icNumber, tableSize):
33      #Initialize Variables
34      icArray = []
35      sumOfChar = 0
36      totalSum = 0
37      constantPrimeNumber = 7
38      hashCode = 0
39      hashValue = 0
40
41      # Firstly, seperating the IC Numbers into 4 parts by slicing
42      part1 = int(icNumber[0:3])
43      part2 = int(icNumber[3:6])
44      part3 = int(icNumber[6:9])
45      part4 = int(icNumber[9:12])
46      icArray.append(part1)
47      icArray.extend([part2, part3, part4])
48
49      # Getting the index of all the numbers in the icArray
50      # Multiplying the ASCII Code of each number with the multiplication between the number and their index
51      for elements in icArray:
52          for char in str(elements):
53              index = str(elements).index(char)
54              sumOfChar += ord(char) * (int(char) * index)
55
56      totalSum += sumOfChar
```

```
45     part4 = int(icNumber[9:12])
46     icArray.append(part1)
47     icArray.extend([part2, part3, part4])
48
49     # Getting the index of all the numbers in the icArray
50     # Multiplying the ASCII Code of each number with the multiplication between the number and their index
51     for elements in icArray:
52         for char in str(elements):
53             index = str(elements).index(char)
54             sumOfChar += ord(char) * (int(char) * index)
55
56         totalSum += sumOfChar
57
58     # Finally get the Hash Code after multiplying a prime number
59     hashCode = (totalSum * constantPrimeNumber)
60
61     # After the modulus operation, we obtain the hash value, which identifies the specific bucket
62     # in the hash table where the data should be placed
63     hashValue = hashCode % tableSize
64
65
66     return hashValue
67
```

Figure 1 & 2: 'hashFuction' function that uses my folding technique

The folding technique used in my hash function is as follows:

1. **Splitting the IC Number:** The 12-digit IC number is divided into four parts, each containing 3 digits. This ensures that the input is processed in smaller, manageable chunks.
2. **Processing Each Part:**
 - For each 3-digit part, each character (digit) is processed individually.
 - The ASCII value of each digit is multiplied by the product of the digit's integer value and its index within the 3-digit part. This adds an element of positional significance to the digits.

3. **Summing and Multiplying:**

- The results of the above multiplications are summed to get a total sum for each part.
- These sums are then combined to form a single total sum, which is further multiplied by a constant prime number (7) to increase the spread of the hash values.

4. **Modulus Operation:**

- The final step applies a modulus operation using the table size. This ensures that the hash value falls within the valid range of bucket indices in the hash table.

By breaking down the IC number and processing each part with a combination of ASCII values and positional multipliers, the folding technique helps in distributing the hash values more evenly across the hash table, reducing the likelihood of collisions.

Given the following hash function:

$$h(k) = k \text{ MOD } 3001$$

Where k is a valid IC Number.

In section (b) of Question 2, we are required to use our answers from 2(a) to write a program that will insert 2000 IC numbers (which can be randomly generated) into two separate hash tables using the functions defined above. The hash function will use an appropriate folding technique and generate the hash code based on the IC number. In the case of a collision, we have to use separate chaining to handle it.

```
# Generate Random IC Numbers
1 usage
def generateRandomIC_Numbers():
    today = datetime.date.today() # Get the current date

    # Generate random year, month, and day
    year = random.randint( a: 0, today.year % 100) # 00 to current year % 100
    month = random.randint( a: 1, b: 12) # 01 to 12
    day = random.randint( a: 1, b: 31) # 01 to 31

    # Create a date string in YYYYMMDD format
    date_str = f"{year:02d}-{month:02d}-{day:02d}"

    # Generate the last 6 digits randomly
    last_6_digits = ''.join(random.choices( population: '0123456789', k=6))

    return date_str + last_6_digits
```

Figure 3: The function that generates random IC Numbers

In Figure 3, the functions generates random IC Numbers to be later inserted into the 2 different hash tables. These IC Numbers are generated so that the month part of the IC only falls between 1 and 12, whereas the day part in the IC falls between 1 to 31.

```
# Given Hash Function
1 usage
def givenHashFunction(icNumber, tableSize):
    return int(icNumber) % tableSize
```

Figure 4: The hash function with the folding technique given in the question

```
def myHashFunction(icNumber, tableSize):  
    #Initialize Variables  
    icArray = []  
    sumOfChar = 0  
    totalSum = 0  
    constantPrimeNumber = 7  
    hashCode = 0  
    hashValue = 0  
  
    # Firstly, seperating the IC Numbers into 4 parts by slicing  
    part1 = int(icNumber[0:3])  
    part2 = int(icNumber[3:6])  
    part3 = int(icNumber[6:9])  
    part4 = int(icNumber[9:12])  
    icArray.append(part1)  
    icArray.extend([part2, part3, part4])  
  
    # Getting the index of all the numbers in the icArray  
    # Multiplying the ASCII Code of each number with the multiplication between the number and their index  
    for elements in icArray:  
        for char in str(elements):  
            index = str(elements).index(char)  
            sumOfChar += ord(char) * (int(char) * index)  
  
        totalSum += sumOfChar  
  
    # Finally get the Hash Code after multiplying a prime number  
    hashCode = (totalSum * constantPrimeNumber)
```



```
# Getting the index of all the numbers in the icArray
# Multiplying the ASCII Code of each number with the multiplication between the number and their index
for elements in icArray:
    for char in str(elements):
        index = str(elements).index(char)
        sumOfChar += ord(char) * (int(char) * index)

    totalSum += sumOfChar

# Finally get the Hash Code after multiplying a prime number
hashCode = (totalSum * constantPrimeNumber)

# After the modulus operation, we obtain the hash value, which identifies the specific bucket
# in the hash table where the data should be placed
hashValue = hashCode % tableSize

return hashValue
```

Figure 5 & 6: The hash function with the folding technique that I have used in Question 2(a)

```
# Insert IC Numbers that are Hashed, into specific Index of the Hash Table
2 usages
def insertInHashTable(hashTable, hashTableSize, icNumber, hashFunction):
    indexNum = hashFunction(icNumber, hashTableSize)

    if hashTable[indexNum] is None:
        hashTable[indexNum] = [icNumber]
    else:
        hashTable[indexNum].append(icNumber)
```

Figure 7: The function that inserts the randomly generated IC Numbers into a Hash Table after passing through a hash function to get the hash value and Bucket Number

```
def main():
    myHashTable = []
    givenHashTable = []
    totalExperiments = 10
    totalIC_NumbersInsertions = 2000
    tableData = []

    for i in range(totalExperiments):
        myHashTable, givenHashTable = experiment()

        myHashTableNumOfCollisions = displayPerformanceResults(myHashTable)
        givenHashTableNumOfCollisions = displayPerformanceResults(givenHashTable)

        myHashTableCollisionRate = (myHashTableNumOfCollisions / totalIC_NumbersInsertions) * 100
        myHashTableCollisionRate = math.ceil(myHashTableCollisionRate * 100) / 100

        givenHashTableCollisionRate = (givenHashTableNumOfCollisions / totalIC_NumbersInsertions) * 100
        givenHashTableCollisionRate = math.ceil(givenHashTableCollisionRate * 100) / 100

        tableData.append([i + 1, myHashTableNumOfCollisions, myHashTableCollisionRate,
                          givenHashTableNumOfCollisions, givenHashTableCollisionRate])

    # Calculate totals and averages
    myHashTableTotalNumberOfCollision = sum(row[1] for row in tableData)
    myHashTableTotalCollisionRate = sum(row[2] for row in tableData)

    givenHashTableTotalNumberOfCollision = sum(row[3] for row in tableData)
    givenHashTableTotalCollisionRate = sum(row[4] for row in tableData)
```

```
myHashTableAverageNumberOfCollisions = myHashTableTotalNumberOfCollision / len(tableData)
myHashTableAverageCollisionRate = myHashTableTotalCollisionRate / len(tableData)

givenHashTableAverageNumberOfCollisions = givenHashTableTotalNumberOfCollision / len(tableData)
givenHashTableAverageCollisionRate = givenHashTableTotalCollisionRate / len(tableData)

# Append totals and averages to the data
tableData.append(['Total', myHashTableTotalNumberOfCollision, myHashTableTotalCollisionRate,
                  givenHashTableTotalNumberOfCollision, givenHashTableTotalCollisionRate])
tableData.append(['Average', myHashTableAverageNumberOfCollisions, myHashTableAverageCollisionRate,
                  givenHashTableAverageNumberOfCollisions, givenHashTableAverageCollisionRate])

# Define headers
headers = ['Experiment', 'Number Of Collisions in My Hash Table', 'Collision Rate in My Hash Table (%)',
           'Number Of Collisions in Given Hash Table', 'Collision Rate in Given Hash Table (%)']

# Print the table
print(tabulate(tableData, headers=headers, tablefmt='grid', stralign='center', numalign="center"))

os.system('pause')
os.system('cls')

if __name__ == "__main__":
    main()
```

Figure 8 & 9:

The code that calculates the performance of both hash functions for 10 times and displays the performance results in table

Experiment	Number Of Collisions in My Hash Table	Collision Rate in My Hash Table (%)	Number Of Collisions in Given Hash Table	Collision Rate in Given Hash Table (%)
1	560	28.01	527	26.35
2	538	26.9	523	26.15
3	561	28.06	533	26.65
4	557	27.85	548	27.4
5	579	28.95	557	27.85
6	562	28.1	534	26.71
7	562	28.1	528	26.4
8	560	28.01	558	27.9
9	538	26.9	545	27.26
10	576	28.8	547	27.35
Total	5593	279.68	5400	270.02
Average	559.3	27.968	540	27.002

Press any key to continue . . .

Figure 10: The table of performance results that is displayed

The results displayed include the number of total collisions and the collision rate of both Hash table for all 10 experiments, it also includes the average of all the 10 experiments. We can see from Figure 10 that the Hash Table using the hash function that uses my folding technique has an average collision rate that is higher than the collision rate of the hash table that uses the given folding technique in Question 2 (b).

Findings and Learnings

1. Collision Rate:

- The average collision rate for the custom hash function was 27.968%, while the average collision rate for the given hash function was 27.002%.
- The custom hash function consistently resulted in a higher number of collisions compared to the given hash function.

2. Performance:

- The given hash function (simple modulus operation) outperformed the custom hash function with respect to collision handling.
- The simpler approach of the given hash function yielded better distribution of IC numbers across the hash table.

3. Conclusion:

- While the custom folding technique added complexity to the hashing process, it did not improve performance and actually led to more collisions.
- The simplicity and efficiency of the given hash function make it more suitable for this application.

Recommendations

Based on the findings, it is recommended to use the given hash function for managing IC numbers in hash tables due to its lower collision rate and better performance. This experiment highlights the importance of evaluating different hashing techniques to ensure efficient data distribution and collision management.

Weaknesses

Several potential weaknesses need to be acknowledged:

1. **Collision Handling:** The use of separate chaining to manage collisions can lead to increased memory usage and slower retrieval times if many collisions occur. Long-linked lists resulting from frequent collisions can degrade search efficiency.
2. **Hash Function Efficiency:** The custom hash function, despite its complexity, has shown a higher collision rate compared to the simpler modulus-based hash function. This suggests that it may not distribute keys as uniformly, affecting overall performance.
3. **Random IC Number Generation:** The method for generating random IC numbers might not accurately represent the real-world distribution, potentially impacting the validity of the performance results.
4. **Performance Metrics:** The current evaluation focuses mainly on collision rates. Additional performance metrics, such as insertion and retrieval times, memory usage, and load factor, could provide a more comprehensive assessment of the system's efficiency.

By addressing these potential weaknesses, future iterations of the system can be optimized for better performance, scalability, and maintainability.

Reflection

Working on this project has provided me with deep insights into the practical applications and intricacies of hash tables, hash functions, and various techniques to manage and optimize them. Through this process, I have learned not only about the theoretical aspects of these data structures but also about their real-world implementation challenges and performance considerations.

Hash Tables and Hash Functions

Implementing the hash table and experimenting with different hash functions highlighted the importance of choosing an appropriate hash function. The hash table's efficiency largely depends on how well the hash function distributes the keys across the available buckets. My initial assumption was that a more complex, custom hash function using a folding technique would perform better. However, the experiments showed that the simpler modulus-based hash function resulted in fewer collisions and better performance.

This experience has taught me that complexity does not always equate to better performance. A good hash function should be simple, efficient, and effective in distributing keys uniformly. The given hash function, despite its simplicity, outperformed the custom function, reinforcing the value of simplicity and proven methods in certain contexts.

Folding Techniques

Developing the custom hash function with a folding technique involved breaking the IC number into parts and applying mathematical operations to generate the hash code. While this approach was intellectually stimulating and provided a deeper understanding of hash function design, it did not yield the expected performance benefits.

This part of the project underscored the need for a balance between innovation and practicality. While experimenting with new techniques is valuable for learning and innovation, it is also essential to benchmark these techniques against established methods to ensure they provide tangible benefits.

Error Handling and Input Validation

Implementing comprehensive error handling and input validation was crucial in ensuring the robustness of the system. By validating the IC numbers and handling potential errors gracefully, I learned the importance of making software resilient to incorrect or unexpected inputs. This aspect of the project reinforced the idea that user input should never be trusted blindly and that thorough validation is key to preventing errors and maintaining data integrity.

Performance Evaluation

The performance evaluation phase, where I compared the collision rates of the two hash functions, provided practical experience in designing and conducting experiments to assess the efficiency of different algorithms. It was enlightening to see how empirical data can challenge assumptions and provide insights that theoretical analysis alone might miss.

Conclusion

In conclusion, this project has been an enriching learning experience, deepening my understanding of hash tables, hash functions, and error handling. I have learned that simplicity can often lead to better performance, the importance of thorough testing and validation, and the need to balance innovation with practicality. These lessons will undoubtedly inform my future work in data structures and algorithms, making me a more effective and thoughtful programmer.

Question 3 - Lee Woei Liang:

Code Implementation

In response to assignment question 3, I have implemented the method of using an adjacency list to display specific adjacent vertices based on the vertex input from the user. The complete code solution, which I have implemented in Python, has been provided in Appendix 3.

```
1  import random
2
3  class Vertex:
4      def __init__(self, name):
5          self.name = name
6          self.edges = {}
7
```

Figure 3.1.1: Class Vertex

Based on figure 3.1.1, I have added a Vertex class to manage vertex attributes. Within the Vertex class, the ‘`__init__`’ method defines name and edges attributes, accepting a parameter for the name value. This structure ensures each vertex is properly instantiated for subsequent graph operations.

```
8  class Graph:
9      def __init__(self):
10         self.vertices = {} # use dictionary to store the vertex and the edges
11         self.adjacency_list = {} # use dictionary to store the to vertex and weight of the edges
12         self.count = 0 #initialize the count to 0
13
14     def add_vertex(self, name):
15         if name not in self.vertices:
16             self.vertices[name] = Vertex(name)
17
```

Figure 3.1.2: Class Graph and add vertex function

Based on the figure 3.1.2, I have included another class named '**Graph**' that was introduced to oversee the graph's attributes, including vertices, adjacency list, and count, all defined within the '**__init__**' method.

For the add vertex function, If a vertex with the given name does not exist in '**self.vertices**', the function creates a new Vertex object with the name call '**Vertex(name)**' and adds it to the '**self.vertices**' dictionary with the key '**name**' .

```

18     def add_edge(self, vertex1, vertex2, weight):
19         if vertex1 not in self.adjacency_list:
20             self.adjacency_list[vertex1] = []
21
22         # Check if the edge already exists
23         edge_exists = False
24         for v, w in self.adjacency_list[vertex1]: # the v and w is the tuples in the adjacency vertex1 list
25             if v == vertex2: #if the vertex of the adjacent vertex1 list is the same as the vertex 2
26                 edge_exists = True
27                 self.count-=1 #decrement the count by 1, have no add the vertex2 and weight to the adjacency list of vertex1 array
28                 break
29
30         if not edge_exists:
31             self.adjacency_list[vertex1].append((vertex2, weight)) # Add the vertex2 and weight to the adjacency list of vertex1 array
32             self.vertices[vertex1].edges[vertex2] = weight # Set the weight of the edge from vertex1 to vertex2 equal to the random weight
33

```

Figure 3.1.3: Add edge function

Based on the '**add_edge**' function shown on figure 3.1.3, it is to add the weight of the edges and the vertex of connected with.

First of all, if the '**vertex1**' is not a key in the '**self.adjacency_list**' then, it adds '**vertex1**' as a key to the dictionary with an empty list as its value. This means that '**vertex1**' is added to the graph as a vertex with no edges

Afterword, set the '**edge_exists = false**' before proceeding to the for loop. The '**for v, w in self.adjacency_list[vertex1]**' is to iterate over each tuple in the adjacency list of '**vertex1**'. Each tuple represents an edge from '**vertex1**' to another vertex, where '**v**' is the vertex it connects to and '**w**' is the weight of the edge. After that, if the '**v**' is the same as the '**vertex2**', then set the '**edge_exists**' to '**True**', to skip the next if statement and minus 1 for the '**count**' variable, then add a break to stop the loop as soon as the edge to be removed is found. This is done for

efficiency, as there's no need to continue checking the rest of the edges once the desired edge is found.

After that, if the `'edge_exists'` is `'False'` then, add the tuple to the `'adjacency_list'` of `'vertex1'` which are the `'vertex2'` and `'weight'`. It represents an edge from vertex1 to vertex2 with a certain weight. For the `'self.vertices[vertex1].edges[vertex2] = weight'`, is setting the weight of the edge from vertex1 to vertex2 to a specific value.

```
34 def list_adjacent_vertex(self, vertex):
35     return self.vertices[vertex].edges #return the specific vertex edges
36
37 def heaviest_vertex(self, vertex):
38     if self.vertices[vertex].edges: # Check if the vertex has any edges
39         return max(self.vertices[vertex].edges, key=self.vertices[vertex].edges.get) # Return the heaviest vertex using max method
40     else:
41         return None # Return None if the vertex has no edges
42
43 def get_adjacent_vertices(self, vertex):
44     return self.adjacency_list.get(vertex, []) # return the adjacent vertices of the specific vertex
45
```

Figure 3.1.4: List adjacent vertex function, heaviest vertex function and get adjacent vertices function

Based on the figure 3.1.4, the function of `'list_adjacent_vertex'` is to return the specific vertex edges.

The `'heaviest_vertex'` function is to return the heaviest vertex within the adjacent vertices. The `'max()'` is a method for getting the maximum value the `'self.vertices[vertex].edges'`. The `'key'` parameter of `'max()'` is a function that takes one argument and returns a `'key'` to use for sorting purposes. The `'self.vertices[vertex].edges.get'` is used, which will return the value for each key in the dictionary, then the max method will compare these values to find the maximum one.

The `'get_adjacent_vertices'` function is to return the specific adjacent vertices based on the vertex value.

```

46 # For creating the 2 graphs
47 def create_graph(vertices, num_edges):
48     graph = Graph()
49
50     # Add the vertices to the graph
51     for vertex in vertices:
52         graph.add_vertex(vertex)
53
54     # Ensure each vertex has one edge
55     for i in range(len(vertices) - 1): # Loop through all vertices except the last one (minus 1 to avoid index out of range error)
56         weight = random.randint(1, 30)
57         graph.add_edge(vertices[i], vertices[i + 1], weight) # Add the edge between the current vertex and the next vertex
58         graph.count += 1
59
60     while graph.count < num_edges: # Loop until reach the required edges number
61         # get random vertex from the vertices list
62         from_vertex = random.choice(vertices)
63         to_vertex = random.choice(vertices)
64
65         weight = random.randint(1, 10)
66         graph.add_edge(from_vertex, to_vertex, weight)
67         graph.count += 1
68     return graph
69

```

Figure 3.1.5: Create graph function

Based on the figure 3.1.5, the *'create_graph'* function adds the vertices and the edges and weight into the graph. Firstly, the for loop is using the parameter of the vertices to declare each of the vertices within the same graph. After that, the another for loop of the *'for i in range(len(vertices) - 1)'* is to add the edges between the *'vertex[i]'* and *'vertex[i+1]'*, and the random weight (from 1 to 30), which have declare inside the for loop. Afterwards, add 1 for the *'graph.count'*. It is used to compare with the parameter of *'num_edges'*, while the *'graph.count'* is less than *'num_edges'* then should loop to add the edges until the *'graph.count'* is not less than *'num_edges'*. Besides, within the while loop, the *'from_vertex'* variable and the *'to_vertex'* variable is using the *'random.choice'* for random picking up the vertices value to set it as the value of the *'from_vertex'* variable and *'to_vertex'* variable.

```

70 # Define vertices
71 vertices1 = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N']
72 vertices2 = ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '15']
73
74 # Create two graph objects
75 graph1 = create_graph(vertices1, 30)
76 graph2 = create_graph(vertices2, 30)

```

Figure 3.1.6: Define the vertices for the graph 1 and graph 2

Based on the figure 3.1.6, the *'vertices1'* and *'vertices2'* is to define each of the vertex values/names. The *graph1* and *graph2* are variables that are assigned the result of the function *'create_graph'*, then pass the vertices and the total number of edges to the function argument.

```
78 # Display the graph 1
79 print("\nGraph 1:")
80 print("Vertices From -> To (Weight)\n")
81 for graph_1_vertex in vertices1:
82     adjacent_vertices = graph1.get_adjacent_vertices(graph_1_vertex)
83     for adj_vertex, weight in adjacent_vertices:
84         print(f"{graph_1_vertex} -> {adj_vertex} ({weight})")
85
86 # Display the graph 2
87 print("\nGraph 2:")
88 print("Vertices From -> To (Weight)\n")
89 for graph_2_vertex in vertices2:
90     adjacent_vertices = graph2.get_adjacent_vertices(graph_2_vertex)
91     for adj_vertex, weight in adjacent_vertices:
92         print(f"{graph_2_vertex} -> {adj_vertex} ({weight})")
93
```

Figure 3.1.7: Display the graph 1 and graph 2

Based on the figure 3.1.7, to define the 2 graphs of the vertices, adjacent vertices and the weight, I have used the same way to initialize the value of the both graphs. First, the for loop is to take the *vertices1*'s vertex to pass in to the *'graph1.get_adjacent_vertices'* function to get each vertices of the adjacent vertex and the weight.

```

84 while True:
85     vertex = input("\nEnter a vertex: ")
86
87     # Check if the vertex exists
88     if vertex in graph1.vertices:
89         adjacent_vertices1 = ' '.join(graph1.list_adjacent_vertex(vertex))
90         heaviest_vertex1 = graph1.heaviest_vertex(vertex)
91         print(f"\nGraph 1 \nVertex {vertex}: Adjacent vertices => {adjacent_vertices1 if adjacent_vertices1 else 'None'}, Heaviest vertex: {heaviest_vertex1 if heaviest_vertex1 is not None else 'None'}")
92     else:
93         print("\nGraph 1 \nError: The Vertex input does not exist in Graph 1.")
94
95     if vertex in graph2.vertices:
96         adjacent_vertices2 = ' '.join(graph2.list_adjacent_vertex(vertex))
97         heaviest_vertex2 = graph2.heaviest_vertex(vertex)
98         print(f"\nGraph 2 \nVertex {vertex}: Adjacent vertices => {adjacent_vertices2 if adjacent_vertices2 else 'None'}, Heaviest vertex: {heaviest_vertex2 if heaviest_vertex2 is not None else 'None'}")
99     else:
100         print("\nGraph 2 \nError: The Vertex input does not exist in Graph 2.")
101
102     # Ask the user if they want to continue
103     continue_query = input("\nDo you want to continue? (Y/N): ")
104     if continue_query.lower() == 'n':
105         break
106     elif continue_query.lower() == 'y':
107         continue
108     else:
109         print("Error: Invalid input. Exiting...")
110         break

```

Figure 3.1.8: Display the specific vertex of the adjacent vertices and the heaviest vertex based on user input

After displaying the 2 Graphs, the system will prompt the user to enter a value for proving the adjacent vertex and the heaviest weight within the adjacent vertex. Based on the figure 3.1.8, i have implemented a while loop when the user enter ‘y’/‘Y’, the system will keep prompting the user to enter a value for checking on the vertex value in the Graph 1 and Graph 2, if the Graph have no the vertex value, then will return an error message where to indicate the vertex input is not existing in the Graph. Can refer to the figure 3.1.8 line 102 and line 109. Where I have used the if else statement for verifying the vertex input value. The if statement condition i have included ‘*vertex in graph1.vertices*’, which is to check is the vertex value existing in the ‘*graph1.vertices*’, if True then will proceed into next line, if the condition is False then will go to the else statement to display the error message. In the if statement, I have called the ‘*join*’ method and the ‘*graph1.list_adjacent_vertex*’ to display the vertex input adjacent vertices. Besides, in the print statement, i have implemented 2 “if and else statement” for filtering if the vertex input have the adjacent vertex then display the value, else will display ‘*None*’ to indicate there have no adjacent vertex for the vertex input, and for the heaviest vertex, i also using the same concept to display the outcome of the heaviest vertex value

Program Outcome

Graph 1: Vertices From -> To (Weight)	Graph 2: Vertices From -> To (Weight)
A -> B (10)	1 -> 2 (25)
A -> F (4)	1 -> 7 (2)
B -> C (10)	1 -> 3 (9)
C -> D (25)	1 -> 12 (8)
C -> N (5)	2 -> 3 (29)
D -> E (12)	2 -> 13 (2)
D -> L (5)	3 -> 4 (23)
D -> H (9)	3 -> 1 (9)
D -> M (10)	4 -> 5 (19)
E -> F (21)	5 -> 6 (11)
F -> G (2)	6 -> 7 (10)
G -> H (28)	6 -> 3 (8)
H -> I (8)	7 -> 8 (8)
H -> K (6)	7 -> 2 (3)
I -> J (27)	8 -> 9 (6)
I -> C (3)	8 -> 4 (7)
J -> K (8)	9 -> 10 (14)
J -> L (1)	10 -> 11 (8)
J -> A (4)	10 -> 14 (7)
J -> G (6)	11 -> 12 (26)
K -> L (14)	11 -> 4 (1)
K -> M (9)	11 -> 5 (6)
L -> M (4)	11 -> 14 (9)
L -> E (10)	12 -> 13 (16)
L -> I (2)	12 -> 5 (10)
M -> N (29)	13 -> 14 (28)
M -> E (7)	14 -> 15 (13)
N -> D (6)	15 -> 3 (4)
N -> A (1)	15 -> 10 (7)
N -> C (4)	15 -> 7 (5)

Figure 3.2.1: Graph 1 and Graph 2 outcome

Based on the figure 3.2.1, the program will show 2 different graphs at the initial, where the first graph will be using Alphabet to represent the vertices value, and the second graph is using the numeric value to represent the vertices value. Besides, each graph will have 30 edges with a random weight (from 1 to 30). The vertices for the first graph are starting from 'A' to 'N', and for the second graph the vertices are starting from '1' to '15'.

```
15 -> 7 (5)

Enter a vertex: 0

Graph 1
Error: The Vertex input does not exist in Graph 1.

Graph 2
Error: The Vertex input does not exist in Graph 2.

Do you want to continue? (Y/N): █
```

Figure 3.2.2: Invalid vertex value

After displaying the 2 graphs, the program will prompt the user to enter a vertex value, it is to get the adjacent vertices of the vertex value and the heaviest vertex from the adjacent vertices. Based on the figure 3.2.2, if the user enters a vertex that is not existing, the program will show an error message for indicating that error. Afterwards, the program will prompt the user to enter whether they want to continue finding the vertex's adjacent vertices and the heaviest vertex. If the user enters 'y', the program will continue. If the user enters 'n', the program will stop. If the input is anything other than these two options, the program will exit.

```
Do you want to continue? (Y/N): y

Enter a vertex: J

Graph 1
Vertex J: Adjacent vertices => K-L-A-G, Heaviest vertex: K

Graph 2
Error: The Vertex input does not exist in Graph 2.

Do you want to continue? (Y/N): █
```

Figure 3.2.3: Valid vertex value for Graph 1

Based on the figure 3.2.3, if the user enters an existing vertex input, then the program will show the related vertex's adjacent vertices and the heaviest vertex within the adjacent vertex. The outcome of this graph program is using the adjacent list method to show to the user. Where can refer to the figure 3.2.3. The Graph 2 will display an error message due to the vertex input not existing in the Graph 2.

```
Do you want to continue? (Y/N): y
Enter a vertex: 15
Graph 1
Error: The Vertex input does not exist in Graph 1.
Graph 2
Vertex 15: Adjacent vertices => 3-10-7, Heaviest vertex: 10
Do you want to continue? (Y/N): n
```

Figure 3.2.4: Valid vertex value for Graph 2

Based on the figure 3.2.4, if the user enters an existing vertex input, then the program will show the related vertex's adjacent vertices and the heaviest vertex within the adjacent vertex. The outcome of this graph program is using the adjacent list method to show to the user. Where can refer to the figure 3.2.4. The Graph 1 will display an error message due to the vertex input not existing in the Graph 1.

Weaknesses

```
69
70 # Define vertices
71 vertices1 = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N']
72 vertices2 = ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '15']
73
74 # Create two graph objects
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
12 -> 7 (9)
12 -> 12 (7)
13 -> 14 (7)
14 -> 15 (11)
14 -> 8 (3)
14 -> 11 (8)

Enter a vertex: 
```

Figure 3.3.1: Graph 2 not using the last element of the ‘*vertice2*’ array

1. Under the Vertices From, the ‘*vertices2*’ sometimes may not use the last element of the array, as shown in Figure 3.3.1.

```
Enter a vertex: 0

Graph 1
Error: The Vertex input does not exist in Graph 1.

Graph 2
Error: The Vertex input does not exist in Graph 2.

Do you want to continue? (Y/N): y

Enter a vertex: J

Graph 1
Vertex J: Adjacent vertices => K-L-A-G, Heaviest vertex: K

Graph 2
Error: The Vertex input does not exist in Graph 2.
```

Figure 3.3.2: Using only one input field to get the result of Graph 1 and Graph 2

2. The program does not have separate input fields for Graph 1 and Graph 2, so when showing the results, both Graph 1 and Graph 2 results will be displayed together, as

shown in Figure 3.3.2. Users will have to parse through both sets of results, which can be inefficient and error-prone.

Reflection

From this project, I have gained a deeper understanding of object-oriented programming by implementing classes and methods to represent and manipulate graph structures. Creating the Vertex and Graph classes improved my skills in designing and implementing classes in Python. The Vertex class encapsulates vertex attributes, while the Graph class manages the overall graph structure, including vertices, edges, and adjacency lists. This modular design makes the code more readable, maintainable, and scalable. Implementing the *'add_edge'* function taught me how to manage edges in a graph, including adding and checking for existing edges efficiently and adding the weight between the 2 vertices. I learned the importance of maintaining data integrity and avoiding redundant entries by carefully checking for existing edges before adding new ones. Through this project, I recognized the importance of robust error handling and input validation. Although the current implementation has minimal error handling, I understand the need for comprehensive checks to ensure the program can gracefully handle unexpected inputs and conditions, for example, "The vertex input was not found in the graph" and "The adjacent vertices are None and the heaviest vertex is None" when the program does not find any value for that vertex input. The process of developing and refining the code highlighted the importance of iterative development. Some of the functions and methods were improved through multiple iterations, which involved identifying weaknesses, testing different approaches, and refining the implementation to achieve better performance and reliability. By implementing a user prompt to input vertices and display their adjacent vertices and heaviest edges, I learned how to design interactive programs that respond to user inputs.

Question 4 - Wong Weng Kean:

Code Implementation & Program Outcome

In section (a) of Question 4, we are required to write a function to calculate and display the housing loan monthly repayment.

```
2 usages  CodeWithSomesh *
def calculateMonthlyMortgage(customerName, principalAmount, yearlyInterestRate, totalLoanYears, results):

    yearlyInterestRatePercentage = yearlyInterestRate/100           # Get the percentage of yearly interest rate
    monthlyInterestRatePercentage = yearlyInterestRatePercentage/12 # Get the percentage of monthly interest rate

    # Calculating the upper part of the Monthly Mortgage Formula
    formulaUpperPart = principalAmount * monthlyInterestRatePercentage
    # Calculating the lower part of the Monthly Mortgage Formula
    formulaLowerPart = 1 - (1/ ((1 + monthlyInterestRatePercentage) ** (12 * totalLoanYears)))

    monthlyMortgage = formulaUpperPart/formulaLowerPart # Calculate monthly mortgage using the complete formula

    # Creating an object for results with customer details
    results[customerName] = {
        "principalAmount": principalAmount,
        "yearlyInterestRate": yearlyInterestRate,
        "tenure": totalLoanYears,
        "monthlyMortgage": f"{monthlyMortgage:.2f}"
    }
```

Figure 1:

The function that calculates and displays the housing loan monthly repayment based on the Youtube Video

In section (b) of Question 4, we are required to write a program to allow users to calculate the monthly repayment for 3 different customers. The program should call the function defined in (a) concurrently. I have added a Menu so that users can choose to either calculate and display the monthly repayment for 3 different customers that are already predefined in the codebase, or to enter the number of customers they want and enter the details they wish to enter.

```
def main():  
  
    while True:  
        print("\nWelcome To Mortgage Calculating System")  
        print("\nAvailable Operations: ")  
        print("1. Calculate the monthly mortgage repayment for 3 predefined customers")  
        print("2. Calculate the monthly mortgage repayment for any number of customers (Enter data manually)")  
        print("3. Exit")  
        choice = input("\nEnter your choice: ")  
        if choice == '1':  
            os.system('cls')  
            print("Calculate the monthly mortgage repayment for 3 predefined customers: ")  
            displayExistingData()  
            os.system('pause')  
            os.system('cls')  
        elif choice == '2':  
            os.system('cls')  
            print("Calculate the monthly mortgage repayment for any number of customers (Enter data manually): ")  
            displayBasedOnUserInput()  
            print()  
            os.system('pause')  
            os.system('cls')  
        elif choice == '3':  
            os.system('cls')  
            print("Exiting the system. Have a good day!")  
            print()  
            os.system('pause')  
            break  
        else:  
            print("Invalid choice. Please enter a number between 1 and 3.")  
            os.system('pause')
```

Figure 2: The Menu of the system

```
for customerName, details in customers.items():
    principalAmount = details["principalAmount"]
    interestRate = details["interestRate"]
    tenure = details["tenure"]

    # Create a thread for each customer
    threads.append(threading.Thread(target=calculateMonthlyMortgage,
                                    args=(customerName, principalAmount, interestRate, tenure, results)))
    threads[-1].start() # Start the thread

# Wait for all threads to finish
for thread in threads:
    thread.join()

displayResults(results)
```

```
numberOfCustomers = getIntInput("\nEnter Number of Customers: ") # Asking customer number

for i in range(numberOfCustomers): # Looping through entered number
    print("-" * 70)
    customerName = input(f'Enter Customer {i + 1} Name: ')
    principalAmount = getFloatInput(f"Enter {customerName}'s Loan Principal Amount: RM")
    interestRate = getFloatInput(f"Enter {customerName}'s Loan Annual Interest Rate (1% to 100%): ")
    tenure = getIntInput(f"Enter {customerName}'s Loan Tenure: ")

    # Create a thread for each customer
    threads.append(threading.Thread(target=calculateMonthlyMortgage,
                                    args=(customerName, principalAmount, interestRate, tenure, results)))
    threads[-1].start() # Start the thread

    print("-" * 70)
    print()

# Wait for all threads to finish
for thread in threads:
    thread.join()

os.system('pause')
os.system('cls')
displayResults(results)
```

Figure 3 & 4:

The functions in the system that applies threading

Concurrent Execution with Multithreading

To ensure that the program calls the `calculateMonthlyMortgage` function concurrently for multiple customers, the following steps were taken:

1. Thread Creation:

- For each customer, a new thread is created with the `threading.Thread` class.
- The target function for each thread is `calculateMonthlyMortgage`, which calculates the monthly repayment based on the provided loan details.
- Each thread is started immediately after creation using the `start()` method.

2. Thread Joining:

- The main program waits for all threads to complete their execution by calling the `join()` method on each thread. This ensures that all calculations are finished before displaying the results.

3. Thread Safety:

- The results dictionary, which stores the calculated repayment details for each customer, is shared among all threads. The design ensures that each thread updates the dictionary safely without causing race conditions.

4. User Input Handling:

- The program offers two modes: calculating repayments for predefined customers and allowing manual input for any number of customers. In both modes, threads are used to handle the calculations concurrently.

This approach demonstrates how multithreading can be utilized to handle concurrent tasks efficiently. However, it is important to note that due to Python's Global Interpreter Lock (GIL), true parallelism is not achieved, but concurrency is sufficient for this I/O-bound task.

By implementing multithreading, the program showcases the practical application of concurrent execution, providing a responsive and efficient solution for calculating housing loan repayments for multiple customers.

Output Observation

Upon running the program, it correctly calculates and displays the monthly mortgage repayments for the given customers. The program provides a user-friendly interface allowing users to input loan details for any number of customers. The results for each customer are displayed in a structured format.

Relevant Theories

Multithreading

The program employs multithreading, a concurrent execution model where multiple threads run independently. Each thread represents a customer and is responsible for calculating and displaying the monthly repayment for one customer. By using multiple threads, the program can perform these calculations for multiple customers simultaneously. However, the performance gain is unlikely to be significant in this scenario and is used mainly as a demonstration of the concept.

Concurrency

Concurrency is the concept of executing multiple tasks concurrently, ensuring progress for each task without waiting for the completion of others. In this application, concurrent threads handle the loan calculations for different customers concurrently.

Thread Safety

When multiple threads access shared resources concurrently, thread safety becomes crucial. In this application, the loan details for each customer are stored in a dictionary, and the `calculateMonthlyMortgage` function is designed to be thread-safe as it operates on individual customer data.

Parallelism

While multithreading provides concurrency, this program may not achieve true parallelism due to Python's Global Interpreter Lock (GIL) and its restrictions on executing multiple bytecode instructions in parallel. However, in scenarios like I/O-bound tasks, multithreading can still offer performance benefits.

Weaknesses

1. Multithreading Issues:

- **GIL (Global Interpreter Lock):** Python's GIL can be a bottleneck in CPU-bound tasks, as it allows only one thread to execute Python bytecode at a time. Although this program is I/O-bound (waiting for user inputs), for CPU-intensive calculations, the GIL would limit the benefits of multithreading.
- **Race Conditions:** If the results dictionary was accessed concurrently by multiple threads, it could lead to race conditions. However, in this program, the results dictionary is modified by the threads in a relatively safe manner. Still, using thread-safe data structures or synchronization mechanisms like locks could be considered for more complex scenarios.

2. Scalability:

- While the current implementation handles a small number of customers well, it may not scale efficiently for a significantly larger number of customers. Managing a large number of threads could lead to overhead and performance degradation.
- To improve scalability, a thread pool or an asynchronous approach could be considered.

Reflection

Developing a program to calculate and display housing loan monthly repayments for multiple customers using multithreading was a valuable exercise in understanding several core programming concepts. This task not only required a solid grasp of financial calculations but also demanded effective implementation of multithreading, concurrency, and thread safety.

Multithreading and Concurrency

The use of multithreading in this project allowed us to run multiple threads concurrently, with each thread handling the mortgage calculation for one customer. This provided a practical demonstration of how concurrency can improve the efficiency of a program by allowing multiple tasks to progress simultaneously. However, the performance gain from multithreading in this particular scenario was minimal due to the nature of the calculations and the constraints of Python's Global Interpreter Lock (GIL).

Thread Safety

Ensuring thread safety was a critical aspect of this project. The mortgage calculation function needed to be designed carefully to avoid race conditions and ensure accurate results. By operating on individual customer data within each thread, we maintained the integrity of the calculations and avoided potential issues that could arise from concurrent access to shared resources.

Challenges and Limitations

One of the main challenges encountered was the potential for race conditions when multiple threads access shared resources concurrently. Although this program handled the `results` dictionary safely, more complex scenarios would require additional synchronization mechanisms. Furthermore, the program's scalability is limited; managing a large number of threads could lead to performance degradation due to the overhead of thread management.

Conclusion

This project was an insightful exploration into the practical applications of multithreading, concurrency, and thread safety. It underscored the importance of careful design and implementation to avoid common pitfalls associated with concurrent programming. While there are areas for improvement, particularly in scalability and platform independence, the program successfully demonstrated the concepts and provided a robust solution for calculating housing loan monthly repayments for multiple customers. This experience will be invaluable in future projects that require efficient and concurrent processing of tasks.

Appendix

Question 1 Code Solution

'smartphone.py' file

```
class Smartphone:
    # Constructor
    # Initializing phones, all nodes originally have all these attributes
    def __init__(self, productCode, brand, model, sellingPrice, colorArray,
quantityOnHandArray, serialNumber):
        self.productCode = productCode
        self.brand = brand
        self.model = model
        self.sellingPrice = sellingPrice
        self.colorArray = colorArray
        self.quantityOnHandArray = quantityOnHandArray
        self.serialNumber = serialNumber

    # The __str__ method is called to get a user-friendly string representation of the
object
    # Particularly useful for debugging and logging
    def __str__(self):
        allColors = ', '.join(self.colorArray)
        allQuantity = ', '.join(self.quantityOnHandArray)

        return (f"Product Code: {self.productCode}, Brand: {self.brand}, Model:
{self.model}, "
                f"Selling Price: RM{self.sellingPrice:.2f}, Serial Number:
{self.serialNumber}"
                f"\nAll Colors: {allColors} \nAll Quantity: {allQuantity}")
```

```
# Updating details of the phone when user select Modify Choice in Menu
def updateDetails(self, newPhoneDetails):
    self.productCode = newPhoneDetails.productCode
    self.brand = newPhoneDetails.brand
    self.model = newPhoneDetails.model
    self.sellingPrice = newPhoneDetails.sellingPrice
    self.colorArray = newPhoneDetails.colorArray
    self.quantityOnHandArray = newPhoneDetails.quantityOnHandArray
    self.serialNumber = newPhoneDetails.serialNumber
    return self
```

'bst.py' file

```
class BST_Node:
    # Constructor
    # Initializing nodes, all nodes originally have no left or right branch
    def __init__(self, phone):
        self.root = phone
        self.left = None
        self.right = None

class BST:
    # Constructor
    # Initializing BST Tree, BST trees originally are always empty with no Root Node
    def __init__(self):
        self.root = None

    def addChild(self, phone, node):
        # Checking to see whether there is already a Root Node
        if self.root is None:
            self.root = BST_Node(phone) # If no Root Node then make the first value
Root Node
        else:
            # If got Root Node, but current product code is BIGGER than Root Node
            if phone.productCode > node.root.productCode:
                # And if there is no RIGHT CHILD for this node, then make the current
phone the RIGHT CHILD
                if node.right is None:
                    node.right = BST_Node(phone)
                # If not run this same method recursively until it becomes a RIGHT
CHILD
            else:
                self.addChild(phone, node.right)
            # If got Root Node, but current product code is SMALLER than Root Node
            elif phone.productCode < node.root.productCode:
                # And if there is no LEFT CHILD for this node, then make the current
phone the LEFT CHILD
                if node.left is None:
                    node.left = BST_Node(phone)
                # If not run this same method recursively until it becomes a LEFT CHILD
            else:
                self.addChild(phone, node.left)
```

```
        else:
            return False

    # Using LVR to return an array of elements (phones)
    def inOrderTraversal(self, node):
        # Initialize empty array
        elements = []

        if node is None:
            return None

        # Visit Left node first, then recursively find for the furthest left leaf node,
        # then add in the array
        if node.left:
            elements += self.inOrderTraversal(node.left)

        # After adding the left node value of the node, then add the root node value
        elements.append(node.root)

        # Lastly add the right node recursively
        if node.right:
            elements += self.inOrderTraversal(node.right)

        # If array is empty then it means no product has been added in the system yet
        if len(elements) == 0:
            return None
        else:
            return elements

    def search(self, productCode, node):

        if node is None:
            return None

        if productCode == node.root.productCode:
            return node.root

        # If the current phone productCode is BIGGER than node's,
        # then run this same method recursively until the phone with given product code
        # is found
        elif productCode > node.root.productCode:
```



```
        return self.search(productCode, node.right)
        # If the current phone productCode is SMALLER than node's,
        # then run this same method recursively until the phone with given product code
is found
        elif productCode < node.root.productCode:
            return self.search(productCode, node.left)
        else:
            return None # Add Print Statement

def modify(self, productCode, newPhoneDetails):
    existingPhone = self.search(productCode, self.root)
    # print(existingPhone)
    modifiedPhone = existingPhone.updateDetails(newPhoneDetails)
    # print(modifiedPhone)
    return modifiedPhone
```

'main.py' file

```
import random
import os
from smartphone import Smartphone
from bst import BST

def getFloatInput(prompt, existingPhone=None):
    while True:
        userInput = input(prompt) or existingPhone.sellingPrice
        try:
            # Try to convert the input to a float
            num = float(userInput)
            return num

        except ValueError:
            # If conversion fails, it's not a valid number
            print("Invalid input. Please enter a valid number.")

def getIntInput(prompt):
    while True:
        userInput = input(prompt)
        if userInput.isdigit():
            return int(userInput)
        else:
            print("Invalid input. Please enter a valid integer number.")

def getYesOrNoInput(prompt):
    while True:
        userInput = input(prompt).upper()
        if userInput == 'Y' or userInput == 'YES':
            return True
        elif userInput == 'N' or userInput == 'NO':
            return False
        else:
            print("Invalid input. Please enter 'Yes' or 'No' only.")
```

```
def addPhone(bst):
    colorArray = []
    quantityArray = []

    productCode = (input("\nEnter Product Code: ")).upper()
    allPhones = bst.inOrderTraversal(bst.root)
    if allPhones is not None:
        for phones in allPhones:
            if productCode == phones.productCode:
                print("\nThere is already a phone under this Product Code.")
                print("Select 1 in the Menu to add a new phone with a different Product Code.")
                print("Select 5 in the Menu to modify the details of the phone with this Product Code.")
            return

    serialNumber = input("Enter Serial Number: ")
    brand = input("Enter Phone Brand: ")
    model = input("Enter Phone Model: ")
    sellingPrice = getFloatInput("Enter Phone Selling Price: RM")

    numberOfColors = getIntInput("Enter Number of Colorways Available: ")
    for i in range(numberOfColors):
        color = input(f"Enter Phone Color {i+1}: ")
        quantityOnHand = getIntInput(f"Enter Quantity of Phone Color {i+1}: ")
        quantityOnHand = str(quantityOnHand)
        colorArray.append(color)
        quantityArray.append(quantityOnHand)

    newPhone = Smartphone(productCode, brand, model, sellingPrice, colorArray, quantityArray, serialNumber)
    bst.addChild(newPhone, bst.root)
    print("\nProduct added successfully.")
    print("\nNew Product Details: ")
    print(newPhone)

def viewAllPhones(bst):
```

```
num = 1
allPhones = bst.inOrderTraversal(bst.root) # Displaying starting from the smallest
Product Code
if allPhones is None:
    print("\nThere aren't any products added in the system")
else:
    for phones in allPhones:
        print(f"\n***** Product {num} ***** ")
        print(f"{phones}")
        num += 1

def searchPhoneByProductCode(bst):
    allPhones = bst.inOrderTraversal(bst.root)
    if allPhones is None:
        print("\nThere aren't any products added in the system")
        return

    displayAvailableProductCodes(bst)

    productCode = (input("\nEnter Product Code to search: ")).upper()
    foundPhone = bst.search(productCode, bst.root)
    print()

    if foundPhone:
        print(foundPhone)
    else:
        print("No phone found for the given Product Code.")

def searchPhonesByBrand(bst):
    foundPhones = []
    num = 1

    allPhones = bst.inOrderTraversal(bst.root)
    if allPhones is None:
        print("\nThere aren't any products added in the system")
        return

    displayAvailableProductBrands(bst)

    brand = input("\nEnter Phone Brand to search: ")
```

```
allPhones = bst.inOrderTraversal(bst.root)

for phones in allPhones:
    if brand.lower() == phones.brand.lower():
        foundPhones.append(phones)

print()
if len(foundPhones) == 0:
    print("No phones found for the given Phone Brand.")
else:
    for phones in foundPhones:
        print(f"\n***** Product {num} ***** ")
        print(f"{phones}")
        num += 1

def modifyPhoneDetails(bst):
    colorArray = []
    quantityArray = []

    allPhones = bst.inOrderTraversal(bst.root)
    if allPhones is None:
        print("\nThere aren't any products added in the system")
        return

    displayAvailableProductCodes(bst)

    productCode = (input("\nEnter Product Code to modify: ")).upper()
    existingPhone = bst.search(productCode, bst.root)
    print()

    if not existingPhone:
        print("Product not found.")
        return

    print("Current phone details:")
    print(existingPhone)
    print()

    print("Enter new details (Leave blank and Press Enter to keep existing value):")
```

```
        serialNumber = input(f"Enter Serial Number ({existingPhone.serialNumber}): ") or
existingPhone.serialNumber
        brand = input(f"Enter Brand ({existingPhone.brand}): ") or existingPhone.brand
        model = input(f"Enter Model ({existingPhone.model}): ") or existingPhone.model
        sellingPrice = getFloatInput(f"Enter Selling Price
(RM{existingPhone.sellingPrice:.2f}): RM", existingPhone)
        # sellingPrice = float(sellingPrice) if sellingPrice else
existingPhone.sellingPrice # Need make changes
        askToModify = getYesOrNoInput("Do you want to modify the Phone Color and Quantity?
(Yes/No): ")

    if askToModify:
        numberOfColors = getIntInput("Enter Number of Colorways Available: ")
        for i in range(numberOfColors):
            color = input(f"Enter Phone Color {i + 1}: ")
            quantityOnHand = getIntInput(f"Enter Quantity of Phone Color {i + 1}: ")
            quantityOnHand = str(quantityOnHand)
            colorArray.append(color)
            quantityArray.append(quantityOnHand)
        else:
            colorArray = existingPhone.colorArray
            quantityArray = existingPhone.quantityOnHandArray

    newPhoneDetails = Smartphone(productCode, brand, model, sellingPrice, colorArray,
quantityArray, serialNumber)
    modifiedPhone = bst.modify(productCode, newPhoneDetails)
    print("\nProduct details updated successfully.")
    print("\nNewly modified phone details:")
    print(modifiedPhone)

def displayAvailableProductCodes(bst):
    num = 1
    allPhones = bst.inOrderTraversal(bst.root) # Displaying starting from the smallest
Product Code
    if allPhones is None:
        print("\nThere aren't any products added in the system")
    else:
```

```
        print(f"\n***** Available Product Codes In The System
***** ")

    for phones in allPhones:
        print(f"{num}. {phones.productCode}")
        num += 1

    print("\n" * 80)

def displayAvailableProductBrands(bst):
    num = 1
    allPhones = bst.inOrderTraversal(bst.root) # Displaying starting from the smallest
    Product Code
    if allPhones is None:
        print("\nThere aren't any products added in the system")
    else:
        print(f"\n***** Available Phone Brands In The System
***** ")

        for phones in allPhones:
            print(f"{num}. {phones.brand}")
            num += 1

        print("\n" * 80)

def menu(bst):
    while True:
        print("\nWelcome To Switch Store Stock Management System")
        print("\nAvailable Operations: ")
        print("1. Create a new product")
        print("2. View all products")
        print("3. Search for a phone by product code")
        print("4. Search for phones by phone brand")
        print("5. Modify phone details")
        print("6. Exit")
        choice = input("\nEnter your choice: ")
        if choice == '1':
            os.system('cls')
            print("Create a new product: ")
```

```
        addPhone(bst)
        print()
        os.system('pause')
        os.system('cls')
    elif choice == '2':
        os.system('cls')
        print("View all products: ")
        viewAllPhones(bst)
        print()
        os.system('pause')
        os.system('cls')
    elif choice == '3':
        os.system('cls')
        print("Search for a phone by product code: ")
        searchPhoneByProductCode(bst)
        print()
        os.system('pause')
        os.system('cls')
    elif choice == '4':
        os.system('cls')
        print("Search for phones by phone brand: ")
        searchPhonesByBrand(bst)
        print()
        os.system('pause')
        os.system('cls')
    elif choice == '5':
        os.system('cls')
        print("Modify phone details: ")
        modifyPhoneDetails(bst)
        print()
        os.system('pause')
        os.system('cls')
    elif choice == '6':
        os.system('cls')
        print("Exiting the system. Have a good day!")
        print()
        os.system('pause')
        break
    else:
        print("Invalid choice. Please enter a number between 1 and 6.")
```



```
        os.system('pause')
        os.system('cls')

def main():
    bst = BST()
    menu(bst)

if __name__ == "__main__":
    main()
```

Question 2 Code Solution

2(a)

```
import os

# Returns user input that has been completely validated
def getNumInput(prompt):
    while True:
        userInput = input(prompt)
        if userInput.isdigit():
            return int(userInput)
        else:
            print("Error: Invalid input. Please enter a valid integer.")

# Returns user input that has been completely validated
def getIcInput(prompt, iteration):
    icNumsArray = []
    num = 1
    while num <= iteration:
        userInput = input(prompt)
        if len(userInput) != 12:
            print("Error: Input must be exactly 12 digits. Please try again.")
            continue
        elif not userInput.isdigit():
            print("Error: Invalid input. Please enter a valid integer.")
            continue

        icNumsArray.append(userInput)

        num += 1

    return icNumsArray

# My Hash Function with my Custom Folding Technique
def hashFunction(icNumber, tableSize):
    #Initialize Variables
    icArray = []
    sumOfChar = 0
    totalSum = 0
```

```
constantPrimeNumber = 7
hashCode = 0
hashValue = 0

# Firstly, seperating the IC Numbers into 4 parts by slicing
part1 = int(icNumber[0:3])
part2 = int(icNumber[3:6])
part3 = int(icNumber[6:9])
part4 = int(icNumber[9:12])
icArray.append(part1)
icArray.extend([part2, part3, part4])

# Getting the index of all the numbers in the icArray
# Multiplying the ASCII Code of each number with the multiplication between the
number and their index
for elements in icArray:
    for char in str(elements):
        index = str(elements).index(char)
        sumOfChar += ord(char) * (int(char) * index)

    totalSum += sumOfChar

# Finally get the Hash Code after multiplying a prime number
hashCode = (totalSum * constantPrimeNumber)

# After the modulus operation, we obtain the hash value, which identifies the
specific bucket
# in the hash table where the data should be placed
hashValue = hashCode % tableSize

return hashValue

if __name__ == "__main__":
    hashCode = 0
    hashCodeArray = []
```

```
numberPrompt = 'How many IC Numbers would you like to enter: '  
num = getNumInput(numberPrompt)  
  
icNumberPrompt = f'Enter IC number without dashes (12 digits): '  
icNums = getIcInput(icNumberPrompt, num)  
  
tableSizePrompt = 'Enter Table Size: '  
tableSize = getNumInput(tableSizePrompt)  
  
for icNum in icNums:  
    hashCode = hashFunction(icNum, tableSize)  
    hashCodeArray.append(hashCode)  
  
for code in hashCodeArray:  
    print(f"Bucket {code} is filled")  
  
print()  
print()  
os.system('pause')  
os.system('cls')
```

2(b)'main1.py' file

The code that calculates the performance of both hash functions for 10 times and displays the performance results in table.

```
import random
import datetime
import os
import math
from tabulate import tabulate

# Generate Random IC Numbers
def generateRandomIC_Numbers():
    today = datetime.date.today() # Get the current date

    # Generate random year, month, and day
    year = random.randint(0, today.year % 100) # 00 to current year % 100
    month = random.randint(1, 12) # 01 to 12
    day = random.randint(1, 31) # 01 to 31

    # Create a date string in YYMMDD format
    date_str = f"{year:02d}{month:02d}{day:02d}"

    # Generate the last 6 digits randomly
    last_6_digits = ''.join(random.choices('0123456789', k=6))

    return date_str + last_6_digits

# My Hash Function
def myHashFunction(icNumber, tableSize):
    #Initialize Variables
    icArray = []
    sumOfChar = 0
    totalSum = 0
    constantPrimeNumber = 7
    hashCode = 0
```

```
hashValue = 0

# Firstly, separating the IC Numbers into 4 parts by slicing
part1 = int(icNumber[0:3])
part2 = int(icNumber[3:6])
part3 = int(icNumber[6:9])
part4 = int(icNumber[9:12])
icArray.append(part1)
icArray.extend([part2, part3, part4])

# Getting the index of all the numbers in the icArray
# Multiplying the ASCII Code of each number with the multiplication between the
number and their index
for elements in icArray:
    for char in str(elements):
        index = str(elements).index(char)
        sumOfChar += ord(char) * (int(char) * index)

    totalSum += sumOfChar

# Finally get the Hash Code after multiplying a prime number
hashCode = (totalSum * constantPrimeNumber)

# After the modulus operation, we obtain the hash value, which identifies the
specific bucket
# in the hash table where the data should be placed
hashValue = hashCode % tableSize

return hashValue

# Given Hash Function
def givenHashFunction(icNumber, tableSize):
    return int(icNumber) % tableSize

# Create Hash Table with the given size
def createHashTable(tableSize):
    return [None] * tableSize
```

```
# Insert IC Numbers that are Hashed, into specific Index of the Hash Table
def insertInHashTable(hashTable, hashTableSize, icNumber, hashFunction):
    indexNum = hashFunction(icNumber, hashTableSize)

    if hashTable[indexNum] is None:
        hashTable[indexNum] = [icNumber]
    else:
        hashTable[indexNum].append(icNumber)

# Display Hash Table Brackets that are Occupied only
def displayHashTable(hashTable):
    num = 1
    print()
    for index in range(len(hashTable)):
        data = hashTable[index]
        if data:
            print(f"{num}. Bucket[{index}]", end="")
            for elements in data:
                print(f" --> {elements}", end="")
            print()
            num += 1

    print()
    print()

# Display Hash Table Performance Results
def displayPerformanceResults(hashTable):
    totalBucketsNum = 0
    emptyBucketNum = 0
    occupiedBucketNum = 0
    bucket_with_1_key = 0
    bucket_with_2_keys = 0
    bucket_with_3_keys = 0
    bucket_with_4_keys = 0
    bucket_with_5_keys = 0
    bucket_with_6_keys = 0
    bucket_with_7_keys = 0
```

```
bucket_with_8_keys = 0
bucket_with_9_keys = 0
bucket_with_10_keys = 0
bucketsWithCollision = 0
numOfCollisions = 0
collisionRate = 0.00

for index in range(len(hashTable)):
    totalBucketsNum += 1
    data = hashTable[index]
    if data:
        if len(data) == 1:
            bucket_with_1_key += 1
        elif len(data) == 2:
            bucket_with_2_keys += 1
        elif len(data) == 3:
            bucket_with_3_keys += 1
        elif len(data) == 4:
            bucket_with_4_keys += 1
        elif len(data) == 5:
            bucket_with_5_keys += 1
        elif len(data) == 6:
            bucket_with_6_keys += 1
        elif len(data) == 7:
            bucket_with_7_keys += 1
        elif len(data) == 8:
            bucket_with_8_keys += 1
        elif len(data) == 9:
            bucket_with_9_keys += 1
        elif len(data) == 10:
            bucket_with_10_keys += 1

        if len(data) > 1:
            numOfCollisions += (len(data) - 1)

    else:
        emptyBucketNum += 1
```



```
occupiedBucketNum = totalBucketsNum - emptyBucketNum
bucketsWithCollision = totalBucketsNum - emptyBucketNum - bucket_with_1_key
collisionRate = (bucketsWithCollision / totalBucketsNum) * 100
collisionRate = math.ceil(collisionRate * 100) / 100

return numOfCollisions

# Display Hash Table Brackets that has Collisions
def displayCollisions(hashTable):
    num = 1
    print()
    for index in range(len(hashTable)):
        data = hashTable[index]
        if data:
            if len(data) > 1:
                print(f"{num}. Bucket[{index}]", end="")
                for elements in data:
                    print(f" --> {elements}", end="")
                print()
            num += 1
    print()
    print()

def experiment():
    # Initialize Variables
    totalIC_Numbers = 2000
    hashTableSize = 3001
    icNumbersArray = []

    # Create 2 Hash Tables
    myHashTable = createHashTable(hashTableSize)
    givenHashTable = createHashTable(hashTableSize)

    # Generate Random IC Numbers and store it in an Array
    for i in range(totalIC_Numbers):
        icNum = generateRandomIC_Numbers()
        icNumbersArray.append(icNum)
```

```
# For every IC Number in the array, turn it into a Hash Code, Find the Index of the
HashCode in the Hash Table,
# And finally Insert the IC Number to the specific index of the Hash Table
# My Hash Table
for icNum in icNumbersArray:
    insertInHashTable(myHashTable, hashTableSize, icNum, myHashFunction)

# Given Hash Table
for icNum in icNumbersArray:
    insertInHashTable(givenHashTable, hashTableSize, icNum, givenHashFunction)

return myHashTable, givenHashTable

def main():
    myHashTable = []
    givenHashTable = []
    totalExperiments = 10
    totalIC_NumbersInsertions = 2000
    tableData = []

    for i in range(totalExperiments):
        myHashTable, givenHashTable = experiment()

        myHashTableNumOfCollisions = displayPerformanceResults(myHashTable)
        givenHashTableNumOfCollisions = displayPerformanceResults(givenHashTable)

        myHashTableCollisionRate = (myHashTableNumOfCollisions /
totalIC_NumbersInsertions) * 100
        myHashTableCollisionRate = math.ceil(myHashTableCollisionRate * 100) / 100

        givenHashTableCollisionRate = (givenHashTableNumOfCollisions /
totalIC_NumbersInsertions) * 100
        givenHashTableCollisionRate = math.ceil(givenHashTableCollisionRate * 100) /
100

        tableData.append([i + 1, myHashTableNumOfCollisions, myHashTableCollisionRate,
givenHashTableNumOfCollisions, givenHashTableCollisionRate])
```

```
# Calculate totals and averages
myHashTableTotalNumberOfCollision = sum(row[1] for row in tableData)
myHashTableTotalCollisionRate = sum(row[2] for row in tableData)

givenHashTableTotalNumberOfCollision = sum(row[3] for row in tableData)
givenHashTableTotalCollisionRate = sum(row[4] for row in tableData)

myHashTableAverageNumberOfCollisions = myHashTableTotalNumberOfCollision /
len(tableData)
myHashTableAverageCollisionRate = myHashTableTotalCollisionRate / len(tableData)

givenHashTableAverageNumberOfCollisions = givenHashTableTotalNumberOfCollision /
len(tableData)
givenHashTableAverageCollisionRate = givenHashTableTotalCollisionRate /
len(tableData)

# Append totals and averages to the data
tableData.append(['Total', myHashTableTotalNumberOfCollision,
myHashTableTotalCollisionRate,
givenHashTableTotalNumberOfCollision,
givenHashTableTotalCollisionRate])
tableData.append(['Average', myHashTableAverageNumberOfCollisions,
myHashTableAverageCollisionRate,
givenHashTableAverageNumberOfCollisions,
givenHashTableAverageCollisionRate])

# Define headers1a
headers = ['Experiment', 'Number Of Collisions in My Hash Table', 'Collision Rate
in My Hash Table (%)',
'Number Of Collisions in Given Hash Table', 'Collision Rate in Given
Hash Table (%)']

# Print the table
print(tabulate(tableData, headers=headers, tablefmt='grid', stralign='center',
numalign="center"))
```

```
os.system('pause')
os.system('cls')

if __name__ == "__main__":
    main()
```

'main2.py' file

The code that calculates the performance of both hash functions for 1 time only and displays the performance results. It also displays the insertions of the IC Numbers into the Buckets, and in the case of a collision, separate chaining is used to handle it.

```
import random
import datetime
import os
import math

# Generate Random IC Numbers
def generateRandomIC_Numbers():
    today = datetime.date.today() # Get the current date

    # Generate random year, month, and day
    year = random.randint(0, today.year % 100) # 00 to current year % 100
    month = random.randint(1, 12) # 01 to 12
    day = random.randint(1, 31) # 01 to 31

    # Create a date string in YYMMDD format
    date_str = f"{year:02d}{month:02d}{day:02d}"

    # Generate the last 6 digits randomly
    last_6_digits = ''.join(random.choices('0123456789', k=6))

    return date_str + last_6_digits
```

```
# My Hash Function
def myHashFunction(icNumber, tableSize):
    #Initialize Variables
    icArray = []
    sumOfChar = 0
    totalSum = 0
    constantPrimeNumber = 7
    hashCode = 0
    hashValue = 0

    # Firstly, seperating the IC Numbers into 4 parts by slicing
    part1 = int(icNumber[0:3])
    part2 = int(icNumber[3:6])
    part3 = int(icNumber[6:9])
    part4 = int(icNumber[9:12])
    icArray.append(part1)
    icArray.extend([part2, part3, part4])

    # Getting the index of all the numbers in the icArray
    # Multiplying the ASCII Code of each number with the multiplication between the
number and their index
    for elements in icArray:
        for char in str(elements):
            index = str(elements).index(char)
            sumOfChar += ord(char) * (int(char) * index)

        totalSum += sumOfChar

    # Finally get the Hash Code after multiplying a prime number
    hashCode = (totalSum * constantPrimeNumber)

    # After the modulus operation, we obtain the hash value, which identifies the
specific bucket
    # in the hash table where the data should be placed
    hashValue = hashCode % tableSize

    return hashValue
```

```
# Given Hash Function
def givenHashFunction(icNumber, tableSize):
    return int(icNumber) % tableSize

# Create Hash Table with the given size
def createHashTable(tableSize):
    return [None] * tableSize

# Insert IC Numbers that are Hashed, into specific Index of the Hash Table
def insertInHashTable(hashTable, hashTableSize, icNumber, hashFunction):
    indexNum = hashFunction(icNumber, hashTableSize)

    if hashTable[indexNum] is None:
        hashTable[indexNum] = [icNumber]
    else:
        hashTable[indexNum].append(icNumber)

# Display Hash Table Brackets that are Occupied only
def displayHashTable(hashTable):
    num = 1
    print()
    for index in range(len(hashTable)):
        data = hashTable[index]
        if data:
            print(f"{num}. Bucket[{index}]", end="")
            for elements in data:
                print(f" --> {elements}", end="")
            print()
            num += 1

    print()
    print()

# Display Hash Table Performance Results
def displayPerformanceResults(hashTable):
    totalBucketsNum = 0
    emptyBucketNum = 0
```

```
occupiedBucketNum = 0
bucket_with_1_key = 0
bucket_with_2_keys = 0
bucket_with_3_keys = 0
bucket_with_4_keys = 0
bucket_with_5_keys = 0
bucket_with_6_keys = 0
bucket_with_7_keys = 0
bucket_with_8_keys = 0
bucket_with_9_keys = 0
bucket_with_10_keys = 0
bucketsWithCollision = 0
numOfCollisions = 0
collisionRate = 0.00

for index in range(len(hashTable)):
    totalBucketsNum += 1
    data = hashTable[index]
    if data:
        if len(data) == 1:
            bucket_with_1_key += 1
        elif len(data) == 2:
            bucket_with_2_keys += 1
        elif len(data) == 3:
            bucket_with_3_keys += 1
        elif len(data) == 4:
            bucket_with_4_keys += 1
        elif len(data) == 5:
            bucket_with_5_keys += 1
        elif len(data) == 6:
            bucket_with_6_keys += 1
        elif len(data) == 7:
            bucket_with_7_keys += 1
        elif len(data) == 8:
            bucket_with_8_keys += 1
        elif len(data) == 9:
            bucket_with_9_keys += 1
        elif len(data) == 10:
            bucket_with_10_keys += 1
```

```
        if len(data) > 1:
            numOfCollisions += (len(data) - 1)

        else:
            emptyBucketNum += 1

occupiedBucketNum = totalBucketsNum - emptyBucketNum
bucketsWithCollision = totalBucketsNum - emptyBucketNum - bucket_with_1_key
collisionRate = (bucketsWithCollision / totalBucketsNum) * 100
collisionRate = math.ceil(collisionRate * 100) / 100

print()
print(f"Total Buckets: {totalBucketsNum}")
print(f"Empty Buckets: {emptyBucketNum}")
print(f"Occupied Buckets: {occupiedBucketNum}")
print(f"Buckets with 1 key: {bucket_with_1_key}")
print(f"Buckets with 2 keys: {bucket_with_2_keys}")
print(f"Buckets with 3 keys: {bucket_with_3_keys}")
print(f"Buckets with 4 keys: {bucket_with_4_keys}")
print(f"Buckets with 5 keys: {bucket_with_5_keys}")
print(f"Buckets with 6 keys: {bucket_with_6_keys}")
print(f"Buckets with 7 keys: {bucket_with_7_keys}")
print(f"Buckets with 8 keys: {bucket_with_8_keys}")
print(f"Buckets with 9 keys: {bucket_with_9_keys}")
print(f"Buckets with 10 keys: {bucket_with_10_keys}")
print(f"Total Number of Collisions: {numOfCollisions}")
print(f"Number of Buckets with Collisions: {bucketsWithCollision}")
print(f"Collision Rate: {collisionRate:.2f}%")
print()
print()

# Display Hash Table Brackets that has Collisions
def displayCollisions(hashTable):
    num = 1
    print()
    for index in range(len(hashTable)):
        data = hashTable[index]
```



```
        if data:
            if len(data) > 1:
                print(f"{num}. Bucket[{index}]", end="")
                for elements in data:
                    print(f" --> {elements}", end="")
                print()
                num += 1

    print()
    print()

def main():
    # Initialize Variables
    totalIC_Numbers = 2000
    hashTableSize = 3001
    icNumbersArray = []

    # Create 2 Hash Tables
    myHashTable = createHashTable(hashTableSize)
    givenHashTable = createHashTable(hashTableSize)

    # Generate Random IC Numbers and store it in an Array
    for i in range(totalIC_Numbers):
        icNum = generateRandomIC_Numbers()
        icNumbersArray.append(icNum)

    # For every IC Number in the array, turn it into a Hash Code, Find the Index of the
    # Hash Code in the Hash Table,
    # And finally Insert the IC Number to the specific index of the Hash Table
    # My Hash Table
    for icNum in icNumbersArray:
        insertInHashTable(myHashTable, hashTableSize, icNum, myHashFunction)

    # Given Hash Table
    for icNum in icNumbersArray:
        insertInHashTable(givenHashTable, hashTableSize, icNum, givenHashFunction)

    # Display My Hash Table Details
```

```

        print("<----- My Hash Table Performance
----->")

displayPerformanceResults(myHashTable)

print("<----- My Hash Table Results ----->")
displayHashTable(myHashTable)

        print("<----- My Hash Table Collisions
----->")
displayCollisions(myHashTable)

        print("<----- My Hash Table Performance
----->")

displayPerformanceResults(myHashTable)

os.system('pause')
os.system('cls')

# Display Given Hash Table Details
        print("<----- Given Hash Table Performance
----->")
displayPerformanceResults(givenHashTable)

        print("<----- Given Hash Table Results
----->")
displayHashTable(givenHashTable)

        print("<----- Given Hash Table Collisions
----->")
displayCollisions(givenHashTable)

        print("<----- Given Hash Table Performance
----->")
displayPerformanceResults(givenHashTable)

os.system('pause')

```

```
os.system('cls')

if __name__ == "__main__":
    main()
```

Question 3 Code Solution

```
import random

class Vertex:
    def __init__(self, name):
        self.name = name
        self.edges = {}

class Graph:
    def __init__(self):
        self.vertices = {} # use dictionary to store the vertice and the
edges
        self.adjacency_list = {} # use dictionary to store the to vertex
and weight of the edges
        self.count = 0 #initialize the count to 0

    def add_vertex(self, name):
        if name not in self.vertices:
            self.vertices[name] = Vertex(name)

    def add_edge(self, vertex1, vertex2, weight):
        if vertex1 not in self.adjacency_list:
            self.adjacency_list[vertex1] = []

        # Check if the edge already exists
        edge_exists = False
        for v, w in self.adjacency_list[vertex1]: # the v and w is the
tuples in the adjacency vertex1 list
            if v == vertex2: #if the vertex of the adjacent vertex1 list
is the same as the vertex 2
                edge_exists = True
                self.count-=1 #decrement the count by 1,have no add the
vertex2 and weight to the adjacency list of vertex1 array
                break

        if not edge_exists:
```

```
        self.adjacency_list[vertex1].append((vertex2, weight)) # Add
the vertex2 and weight to the adjacency list of vertex1 array
        self.vertices[vertex1].edges[vertex2] = weight # Set the
weight of the edge from vertex1 to vertex2 equal to the random weight

    def list_adjacent_vertex(self, vertex):
        return self.vertices[vertex].edges #return the specific vertex
edges

    def heaviest_vertex(self, vertex):
        if self.vertices[vertex].edges: # Check if the vertex has any
edges
            return max(self.vertices[vertex].edges,
key=self.vertices[vertex].edges.get) # Return the heaviest vertex using
max method
        else:
            return None # Return None if the vertex has no edges

    def get_adjacent_vertices(self, vertex):
        return self.adjacency_list.get(vertex, []) # return the adjacent
vertices of the specific vertex

# For creating the 2 graphs
def create_graph(vertices, num_edges):
    graph = Graph()

    # Add the vertices to the graph
    for vertex in vertices:
        graph.add_vertex(vertex)

    # Ensure each vertex has one edge
    for i in range(len(vertices) -1): # Loop through all vertices except
the last one (minus 1 to avoid index out of range error)
        weight = random.randint(1, 30)
        graph.add_edge(vertices[i], vertices[i + 1], weight) # Add the
edge between the current vertex and the next vertex
    graph.count += 1
```

```
        while graph.count < num_edges: # Loop until reach the required edges
number
            # get random vertex from the vertices list
            from_vertex = random.choice(vertices)
            to_vertex = random.choice(vertices)

            weight = random.randint(1, 10)
            graph.add_edge(from_vertex, to_vertex, weight)
            graph.count += 1
        return graph

# Define vertices
vertices1 = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
'M', 'N']
vertices2 = [ '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11',
'12', '13', '14', '15']

# Create two graph objects
graph1 = create_graph(vertices1, 30)
graph2 = create_graph(vertices2, 30)

# Display the graph 1
print("\nGraph 1:")
print("Vertices From -> To (Weight)\n")
for graph_1_vertex in vertices1:
    adjacent_vertices = graph1.get_adjacent_vertices(graph_1_vertex)
    for adj_vertex, weight in adjacent_vertices:
        print(f"{graph_1_vertex} -> {adj_vertex} ({weight})")

# Display the graph 2
print("\nGraph 2:")
print("Vertices From -> To (Weight)\n")
for graph_2_vertex in vertices2:
    adjacent_vertices = graph2.get_adjacent_vertices(graph_2_vertex)
    for adj_vertex, weight in adjacent_vertices:
        print(f"{graph_2_vertex} -> {adj_vertex} ({weight})")

while True:
```

```
vertex = input("\nEnter a vertex: ")

# Check if the vertex exists
if vertex in graph1.vertices:
    adjacent_vertices1 = '-'.join(graph1.list_adjacent_vertex(vertex))
    heaviest_vertex1 = graph1.heaviest_vertex(vertex)
    print(f"\nGraph 1 \nVertex {vertex}: Adjacent vertices =>
{adjacent_vertices1 if adjacent_vertices1 else 'None'}, Heaviest vertex:
{heaviest_vertex1 if heaviest_vertex1 is not None else 'None'}")
else:
    print("\nGraph 1 \nError: The Vertex input does not exist in Graph
1.")

if vertex in graph2.vertices:
    adjacent_vertices2 = '-'.join(graph2.list_adjacent_vertex(vertex))
    heaviest_vertex2 = graph2.heaviest_vertex(vertex)
    print(f"\nGraph 2 \nVertex {vertex}: Adjacent vertices =>
{adjacent_vertices2 if adjacent_vertices2 else 'None'}, Heaviest vertex:
{heaviest_vertex2 if heaviest_vertex2 is not None else 'None'}")
else:
    print("\nGraph 2 \nError: The Vertex input does not exist in Graph
2.")

# Ask the user if they want to continue
continue_query = input("\nDo you want to continue? (Y/N): ")
if continue_query.lower() == 'n':
    break
elif continue_query.lower() == 'y':
    continue
else:
    print("Error: Invalid input. Exisitng...")
    break
```

Question 4 Code Solution

```
import threading
import os

# Function that ensures the inputs are float variable and more than 0
def getFloatInput(prompt):
    while True:
        userInput = input(prompt)
        try:
            # Try to convert the input to a float
            num = float(userInput)

            if num < 1:
                print("Invalid input. Please enter a number that is at least 1 or higher.\n")
                continue

            return num

        except ValueError:
            # If conversion fails, it's not a valid number
            print("Invalid input. Please enter a valid number.\n")

# Function that ensures the inputs are int variable and more than 0
def getIntInput(prompt):
    while True:
        userInput = input(prompt)
        if userInput.isdigit():
            return int(userInput)
        else:
            print("Invalid input. Please enter a valid and positive integer number.\n")

def calculateMonthlyMortgage(customerName, principalAmount, yearlyInterestRate, totalLoanYears, results):
    yearlyInterestRatePercentage = yearlyInterestRate/100 # Get the percentage of yearly interest rate
```



```
monthlyInterestRatePercentage = yearlyInterestRatePercentage/12      # Get the
percentage of monthly interest rate

# Calculating the upper part of the Monthly Mortgage Formula
formulaUpperPart = principalAmount * monthlyInterestRatePercentage
# Calculating the lower part of the Monthly Mortgage Formula
formulaLowerPart = 1 - (1/ ((1 + monthlyInterestRatePercentage) ** (12 *
totalLoanYears)))

monthlyMortgage = formulaUpperPart/formulaLowerPart # Calculate monthly mortgage
using the complete formula
results[customerName] = {
    "principalAmount": principalAmount,
    "yearlyInterestRate": yearlyInterestRate,
    "tenure": totalLoanYears,
    "monthlyMortgage": f"{monthlyMortgage:.2f}"
}

def displayResults(results):
    print()
    for customerName, details in results.items():
        principalAmount = details["principalAmount"]
        interestRate = details["yearlyInterestRate"]
        tenure = details["tenure"]
        mortgageAmount = details["monthlyMortgage"]

        print("*" * 60)
        print(f"Customer Name: {customerName}")
        print(f"Principal Amount: RM{principalAmount}")
        print(f"Interest Rate: {interestRate}%")
        print(f"Loan Tenure: {tenure} Years")

        print("-" * 45)
        print(f"{customerName}'s monthly mortgage is: RM{mortgageAmount}")
        print("-" * 45)

    print("*" * 60)
    print()
```

```
def displayExistingData():  
    # Initialize Variables  
    results = {}  
    threads = []  
  
    # Define the loan details for the 3 customers  
    customers = {  
        "Somesh": {  
            "principalAmount": 100000,  
            "interestRate": 4.5,  
            "tenure": 15  
        },  
        "Weng Kean": {  
            "principalAmount": 200000,  
            "interestRate": 4.5,  
            "tenure": 30  
        },  
        "Jun Khai": {  
            "principalAmount": 300000,  
            "interestRate": 4.5,  
            "tenure": 20  
        }  
    }  
  
    # Create and start a thread for each customer  
  
    for customerName, details in customers.items():  
        principalAmount = details["principalAmount"]  
        interestRate = details["interestRate"]  
        tenure = details["tenure"]  
  
        # Create a thread for each customer  
        threads.append(threading.Thread(target=calculateMonthlyMortgage,  
                                         args=(customerName, principalAmount,  
                                               interestRate, tenure, results)))  
        threads[-1].start() # Start the thread  
  
    # Wait for all threads to finish
```

```
        for thread in threads:
            thread.join()

        displayResults(results)

def displayBasedOnUserInput():
    # Initialize Variables
    results = {}
    threads = []

    numberOfCustomers = getIntInput("\nEnter Number of Customers: ") # Asking customer
number

    for i in range(numberOfCustomers): # Looping through entered number
        print("-" * 70)
        customerName = input(f'Enter Customer {i + 1} Name: ')
        principalAmount = getFloatInput(f"Enter {customerName}'s Loan Principal Amount:
RM")
        interestRate = getFloatInput(f"Enter {customerName}'s Loan Annual Interest Rate
(1% to 100%): ")
        tenure = getIntInput(f"Enter {customerName}'s Loan Tenure: ")

        # Create a thread for each customer
        threads.append(threading.Thread(target=calculateMonthlyMortgage,
                                         args=(customerName, principalAmount,
interestRate, tenure, results)))
        threads[-1].start() # Start the thread

        print("-" * 70)
        print()

    # Wait for all threads to finish
    for thread in threads:
        thread.join()

    os.system('pause')
    os.system('cls')
    displayResults(results)
```

```
def main():
    results = {}

    while True:
        print("\nWelcome To Mortgage Calculating System")
        print("\nAvailable Operations: ")
        print("1. Calculate the monthly mortgage repayment for 3 predefined customers")
        print("2. Calculate the monthly mortgage repayment for any number of customers")
        print("(Enter data manually)")
        print("3. Exit")
        choice = input("\nEnter your choice: ")
        if choice == '1':
            os.system('cls')
            print("Calculate the monthly mortgage repayment for 3 predefined customers:")
            displayExistingData()
            os.system('pause')
            os.system('cls')
        elif choice == '2':
            os.system('cls')
            print("Calculate the monthly mortgage repayment for any number of customers")
            print("(Enter data manually): ")
            displayBasedOnUserInput()
            print()
            os.system('pause')
            os.system('cls')
        elif choice == '3':
            os.system('cls')
            print("Exiting the system. Have a good day!")
            print()
            os.system('pause')
            break
        else:
            print("Invalid choice. Please enter a number between 1 and 3.")
            os.system('pause')
            os.system('cls')
```

```
if __name__ == "__main__":  
    main()
```