**Coursework Cover Sheet**

**Section A - To be completed by the student**

|  |  |
| --- | --- |
| Full Name:  **Tan Yi Jia** | |
| CU Student ID Number:  **12672752** | |
| Semester:  **5** | |
| Lecturer:  **Koo Lee Chun** | |
| Module Code and Title:  **5003CEM Advanced Algorithms** | |
| Assignment No. / Title:  **Coursework** | % of Module Mark  **67%** |
| Hand out date:  **20 April 2022** | Due date:  **9 JUNE 2023** |
| 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 the undersigned confirm that I have read and agree to abide by the University regulations on plagiarism and cheating and Faculty coursework policies and procedures. I confirm that this piece of work is my own. I consent to appropriate storage of my work for plagiarism checking.  Signature(s): yijia | |

**Section B - To be completed by the module leader**

|  |  |  |
| --- | --- | --- |
| Intended learning outcomes assessed by this work:  LO1: Understand and select appropriate algorithms for solving a range of problems and reason about their complexity and efficiency.  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.  LO4: Describe the issue of data consistency in non-synchronous applications.  LO5. Design and implement a basic concurrent application. | | |
| Marking scheme | Max | Mark |
|  |  |  |
| Total |  |  |

|  |
| --- |
| Lecturer’s Feedback |
| Internal Moderator’s Feedback |

Contents

[Programming Tasks 4](#_Toc137033115)

[Question 1: Cake ordering System 4](#_Toc137033116)

[Output 4](#_Toc137033117)

[Weakness of solution 34](#_Toc137033118)

[Code References 34](#_Toc137033119)

[Question 2: Hash Table Collision technique average cost comparison 35](#_Toc137033120)

[Output 36](#_Toc137033121)

[Discussion 37](#_Toc137033122)

[Weakness of solution 38](#_Toc137033123)

[Code References 38](#_Toc137033124)

[Question 3: Graph 39](#_Toc137033125)

[Output 39](#_Toc137033126)

[Discussion 41](#_Toc137033127)

[Weakness of solution 42](#_Toc137033128)

[Code References 42](#_Toc137033129)

[Question 4: Concurrent process 43](#_Toc137033130)

[Output 43](#_Toc137033131)

[Discussion 45](#_Toc137033132)

[Weakness of solution 46](#_Toc137033133)

[Code References 46](#_Toc137033134)

[Reflection 47](#_Toc137033135)

[References 48](#_Toc137033136)

[Appendix 49](#_Toc137033137)

[Question 1 49](#_Toc137033138)

[Question 2 60](#_Toc137033139)

[Question 3 62](#_Toc137033140)

[Question 4 64](#_Toc137033141)

# Programming Tasks

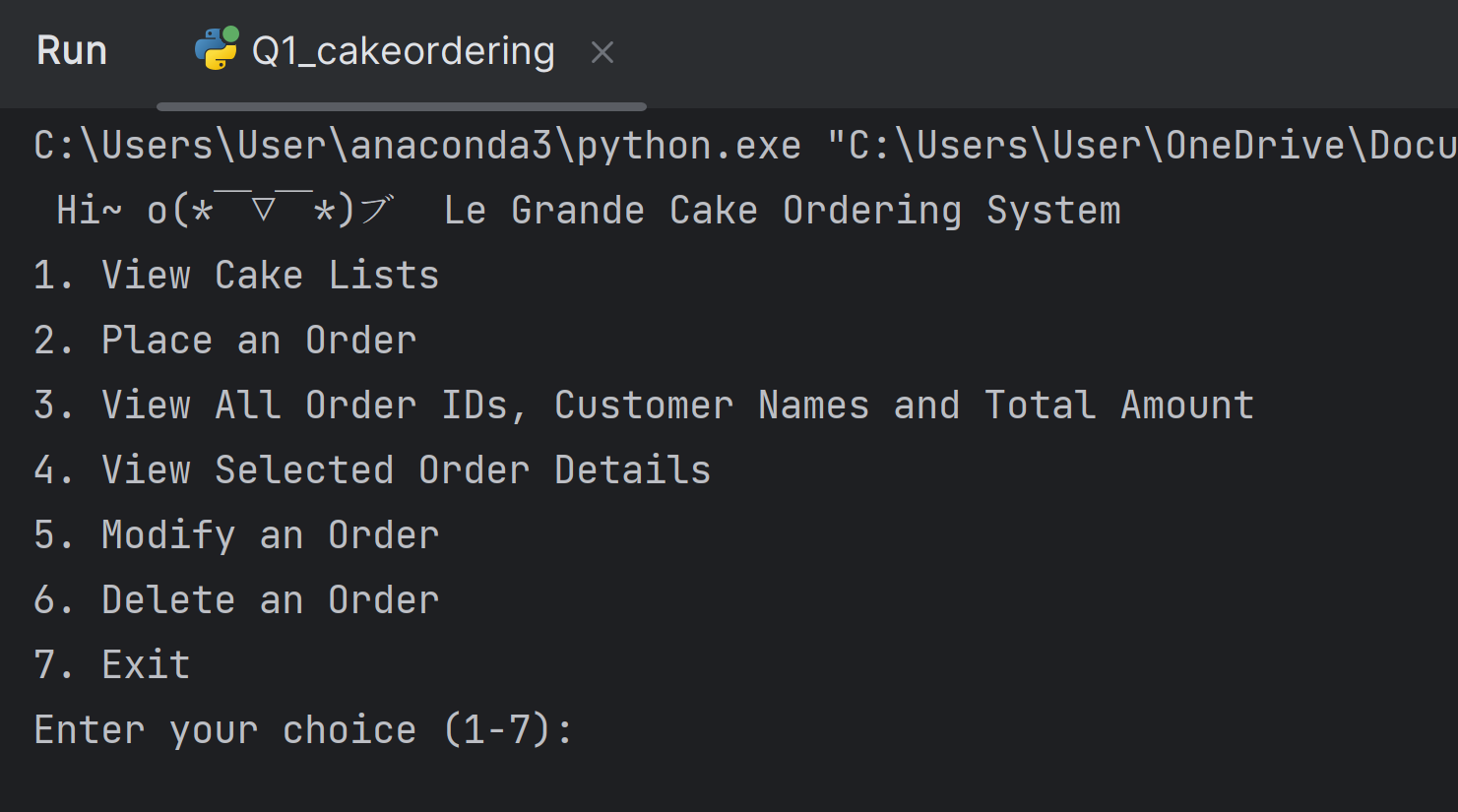
## Question 1: Cake ordering System

Design and develop a Cake ordering system based on following requirements:

* The program should be menu driven, giving the user various choices of operations that allows a user to place an order for the cake(s), view list of cake order details, modify or delete a particular order if necessary.
* For every cake order, the following information will be stored:
* Order ID (Auto assigned, no duplication), Cake Code, flavour (eg: Strawberry, chocolate), weight (Kg), Unit Price, Qty and customer information who order the cake.
* The customer information consists of customer ID, name, address and contact number.
* The system shall display additional information that is amount (unit price \* qty) when viewing the order details.
* The program must use BST data structure to facilitate each operation.
* The system shall demonstrate a good OOP design, data validation and error handling.

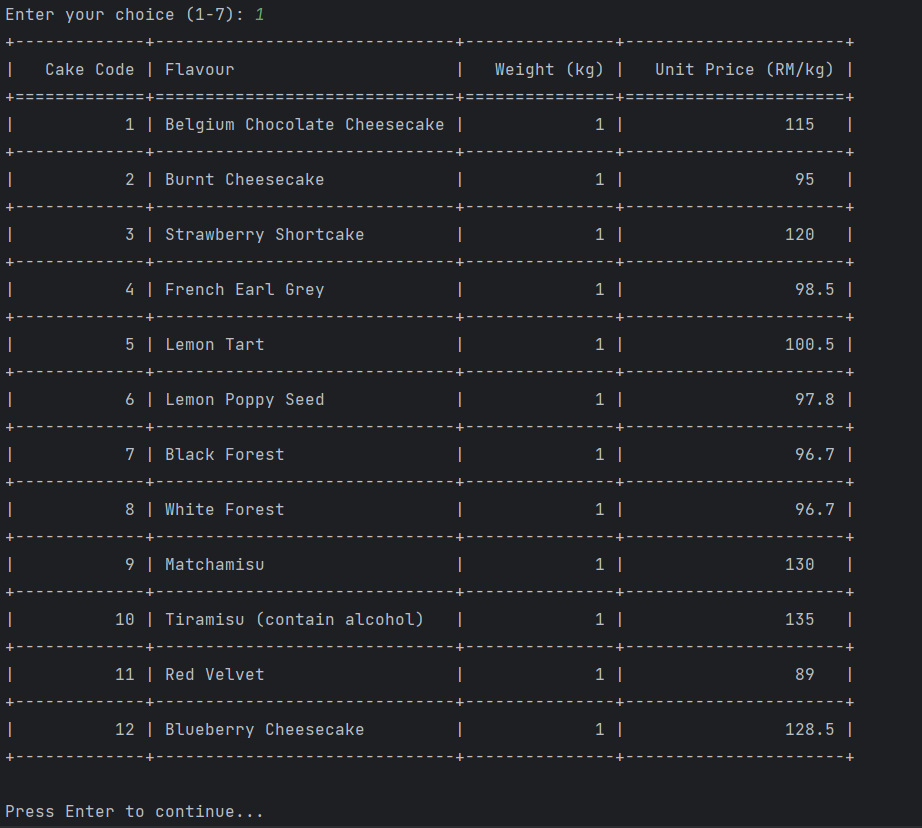
### Output

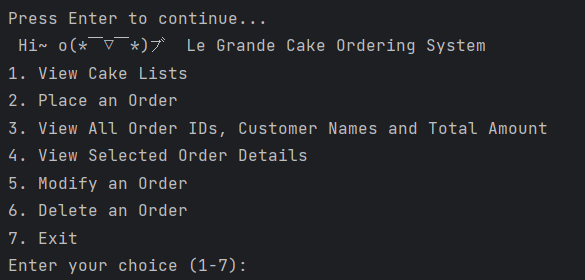
1. Starting Menu. Prompt user to input which function the user choose to do.



1. When user input “1” (view cake lists)

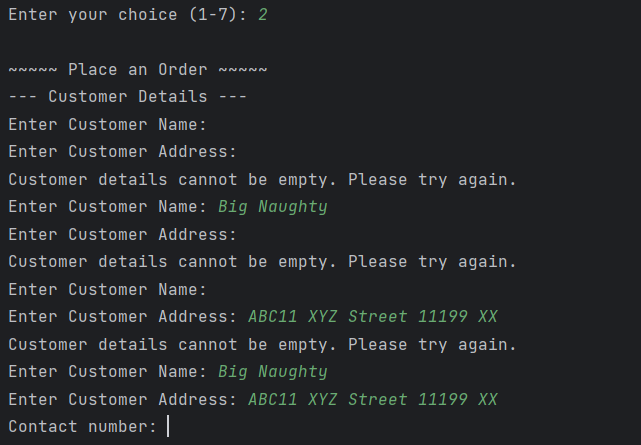
The cake lists showing the cake code, flavour, weight, and unit price will print out in a table for the user to review. If the user wanted to return to the starting menu, then press ‘Enter’ to return. The output is shown in the second picture below.



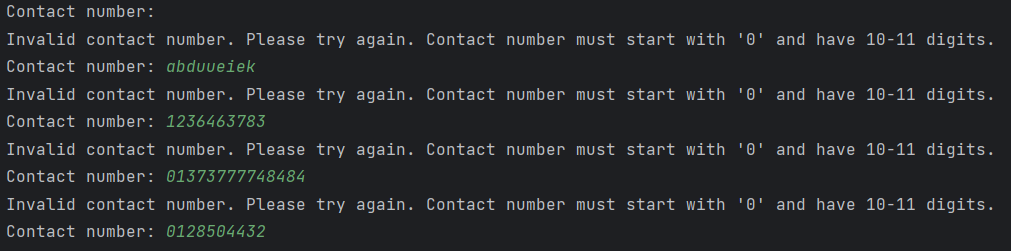


1. When user input “2” (place an order)

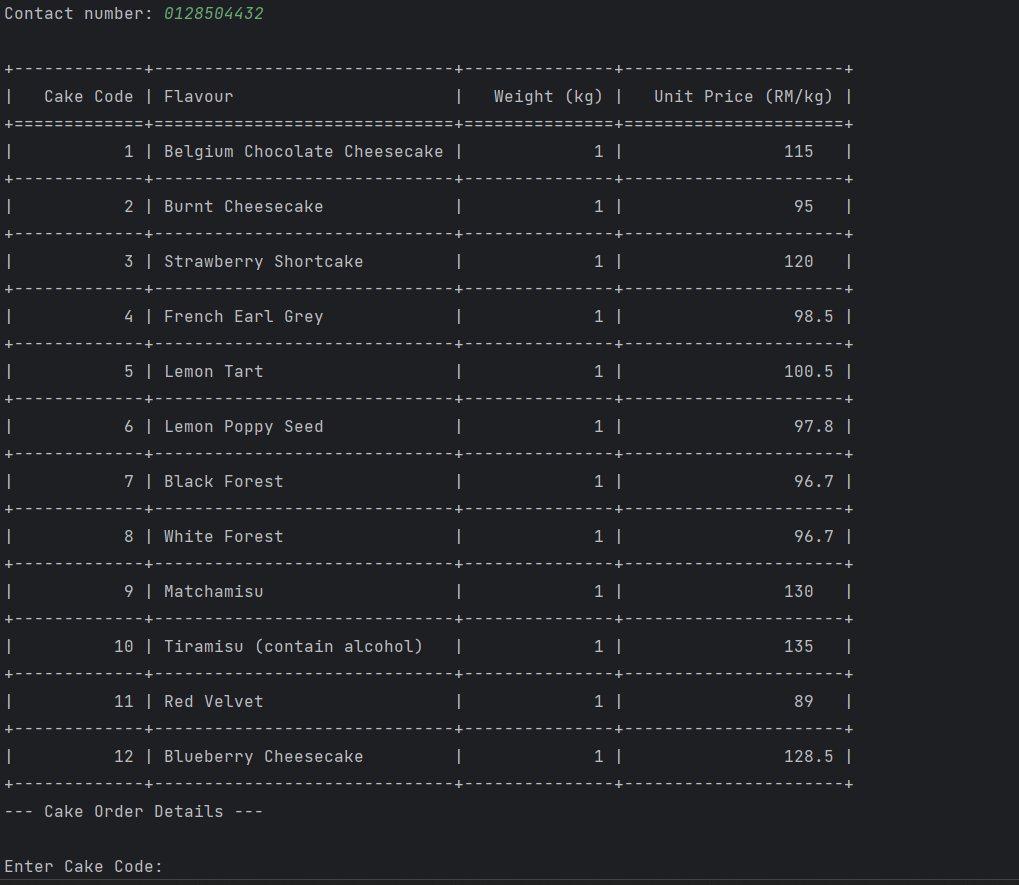
Firstly, the user is required to enter the customer's details. If the name or address is empty, the programme will loop until the user enters the name and details. After the name and address are entered, the programme will only ask for the user to enter the contact number.



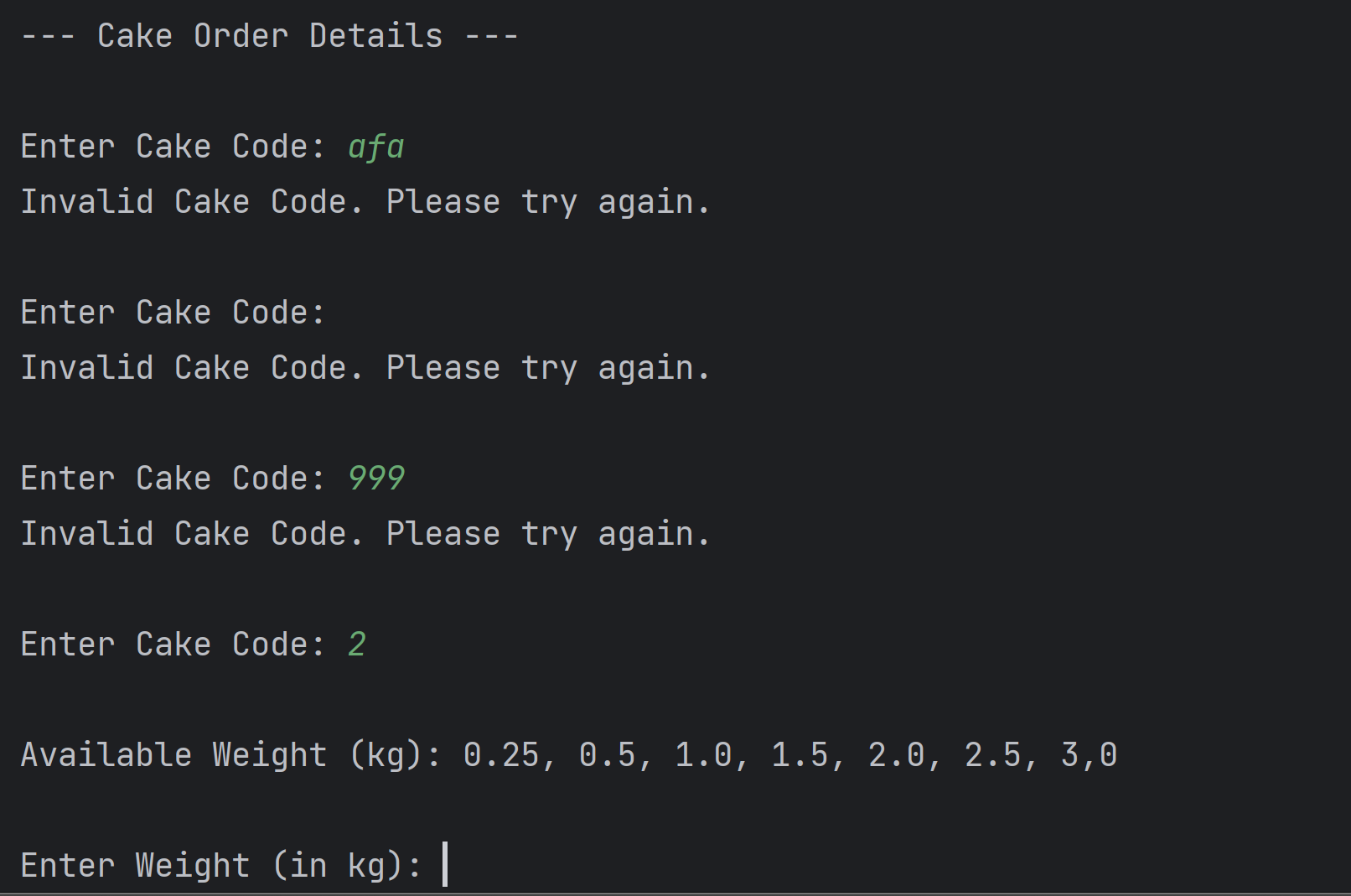
The programme will validate that the contact number must start with 0 and include 10–11 digits before proceeding to the next.



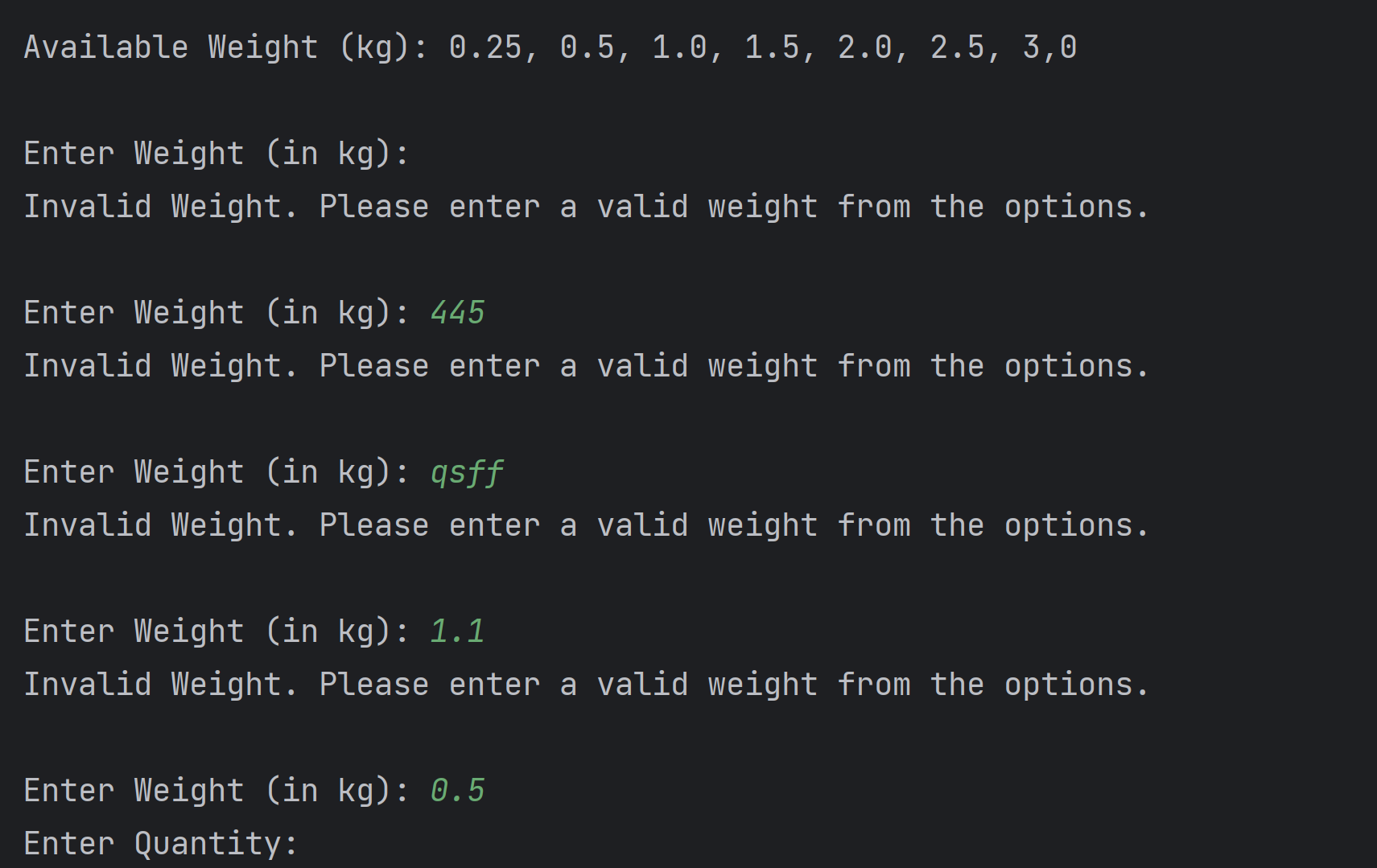
The cake lists will be shown to the user as a reference so the user can place the cake order easily. The programme will ask for user input for the cake code to place the order.



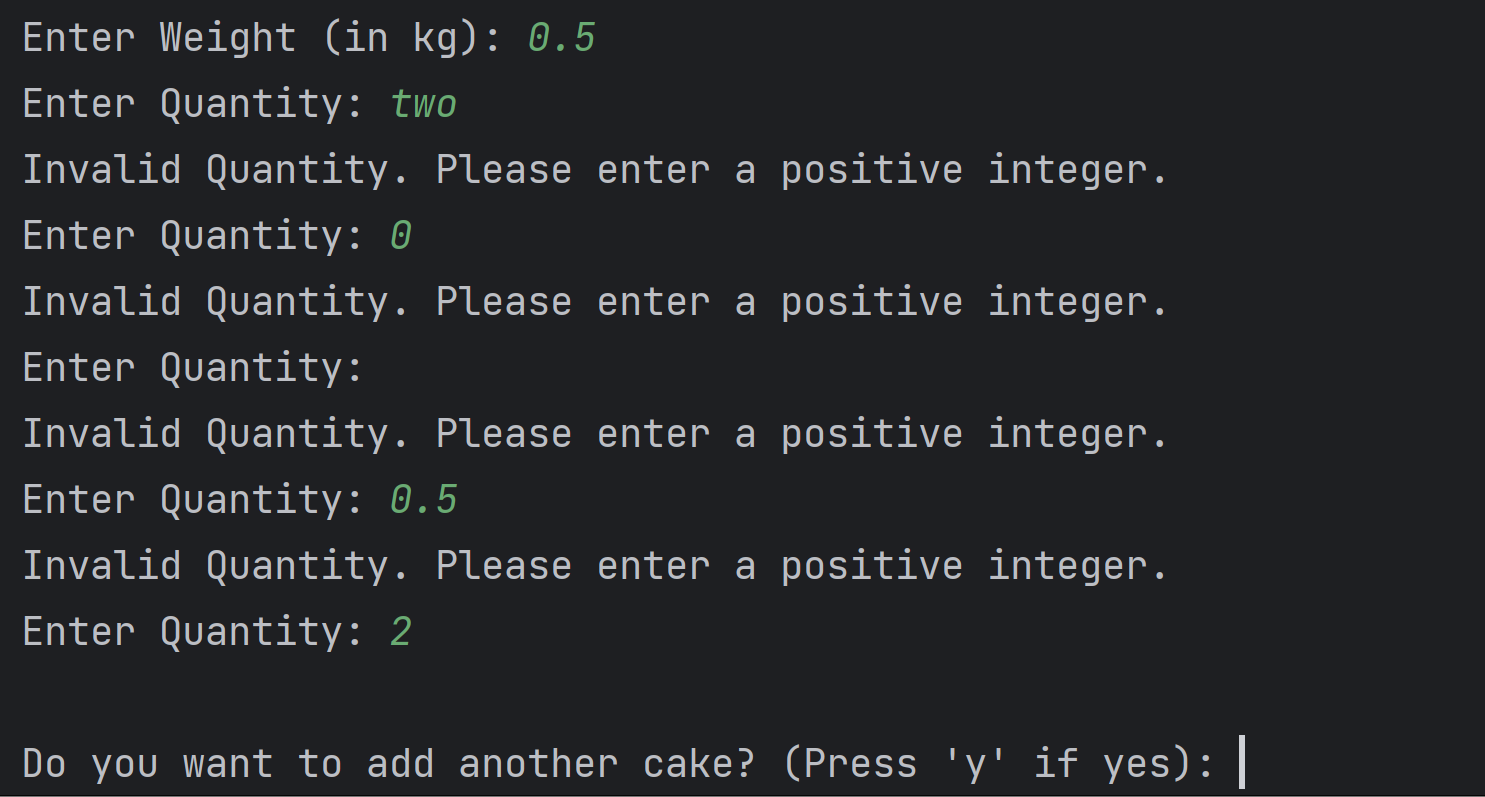
If the cake code entered is not valid, the programme will show the message "Invalid Cake Code. Please try again.". After entering a valid cake code (1–12), it will show the weight available for the cake and ask for input.



There is validation in the user input for the weight that the input weight must be in the available weight lists. Otherwise, the system will show the message "Invalid Weight. Please enter a valid weight from the options.". If the weight is available, the user is prompted to enter the quantity.

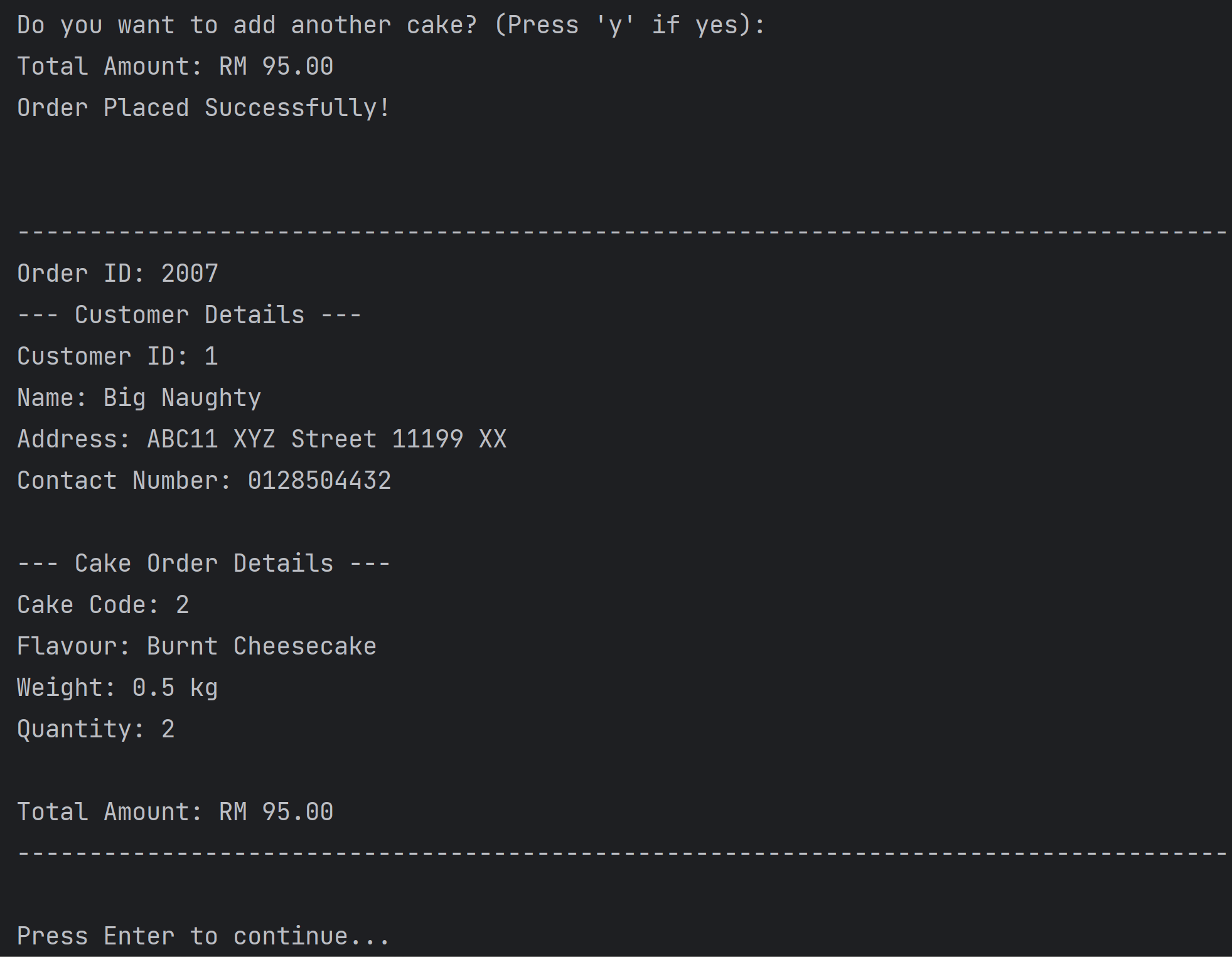


The quantity must be an integer that is bigger than 0, otherwise the error will print and ask for input until a valid integer is entered. Next, the system will ask the user if they want to add more cakes to the order or not.

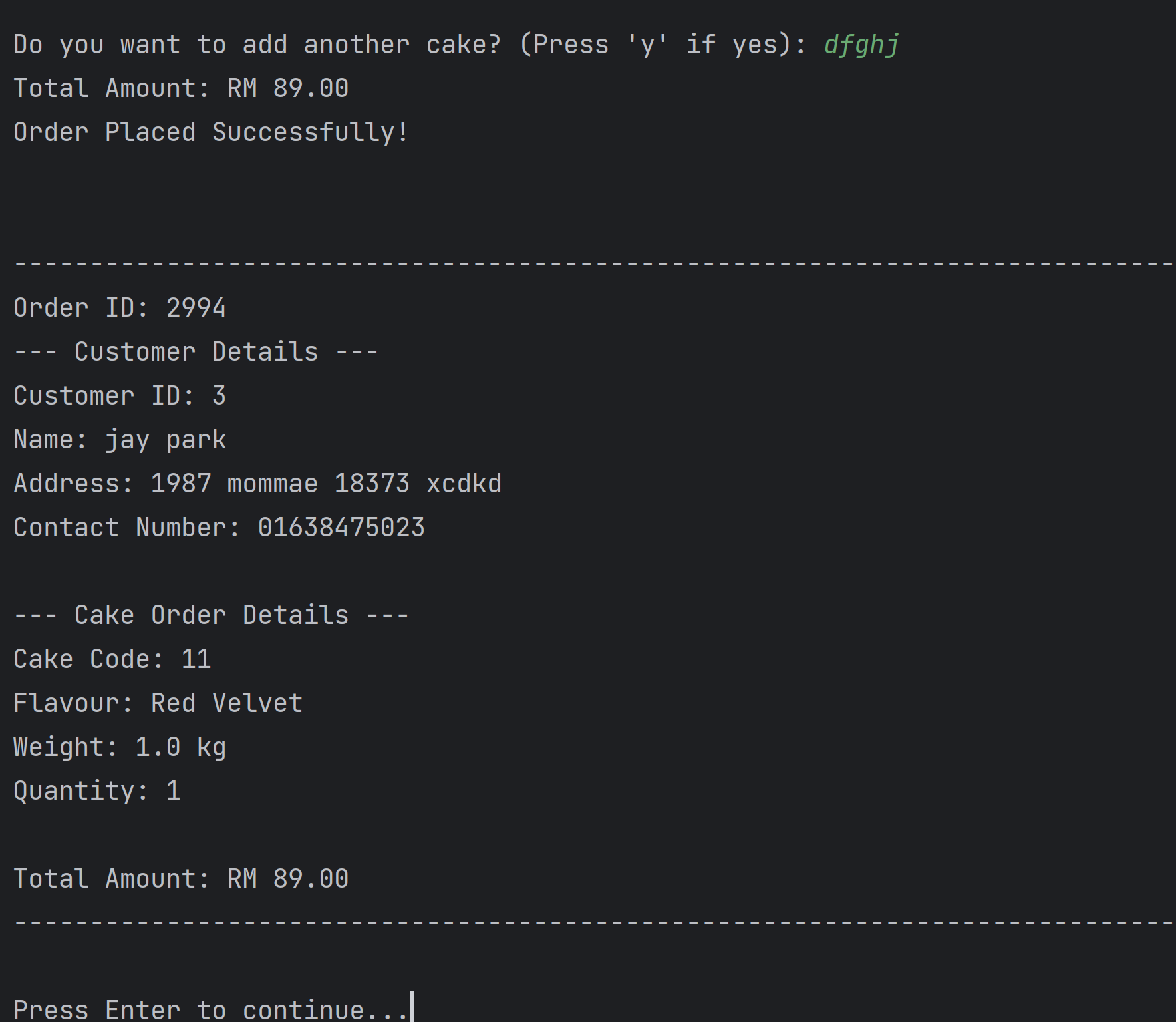


The validation on the input for asking the user whether he or she wants to add more cakes will check on the input whether it is ‘y’ or "Y." Any input, such as ‘enter’, any integer, string, etc. other than ‘y," will be considered as do not add more cake. If an input other than ‘y" is entered, the order will be inserted into the BST, and the order details will print out as shown in the image below.

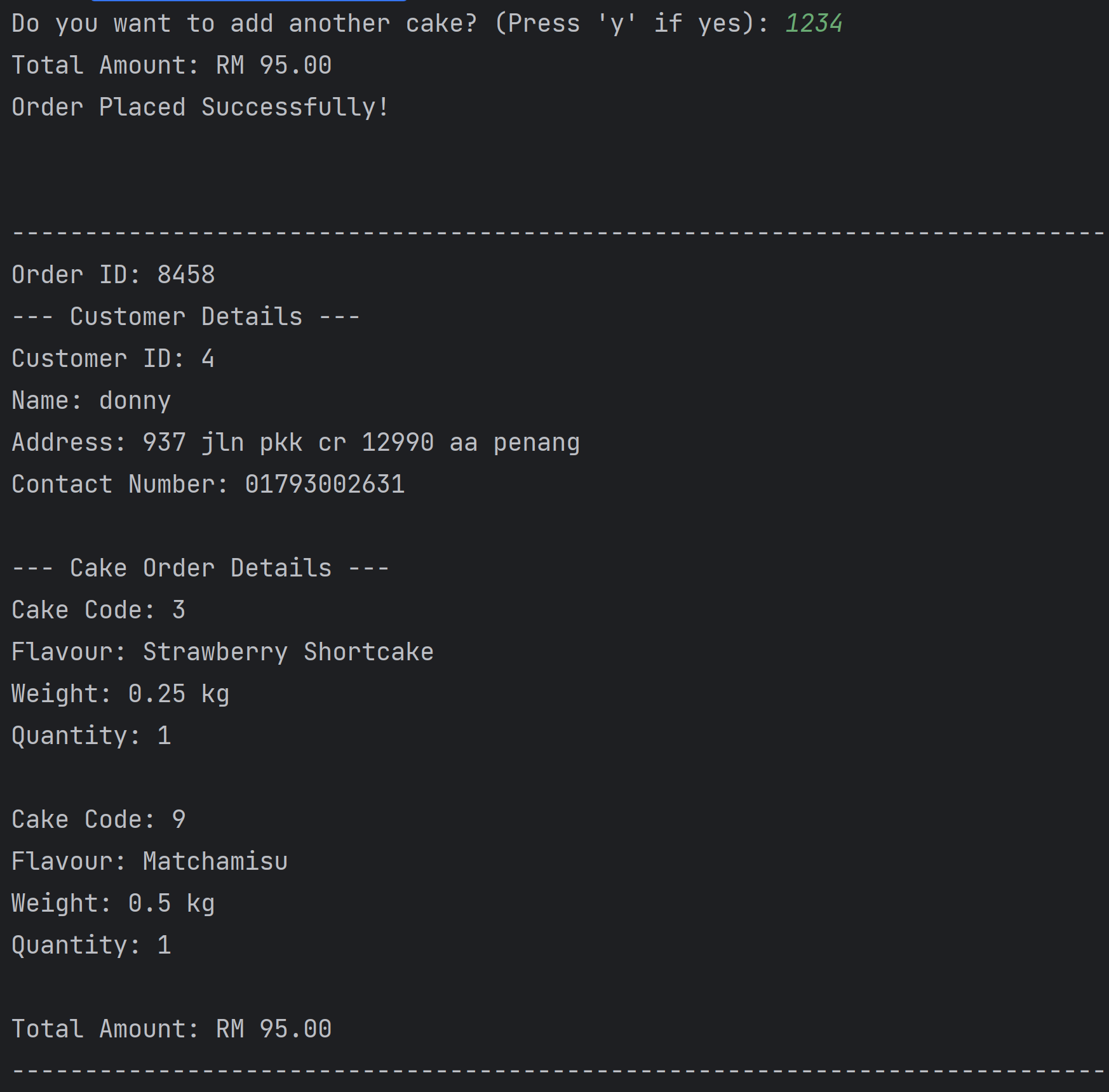
Example 1: ‘Enter’ is pressed



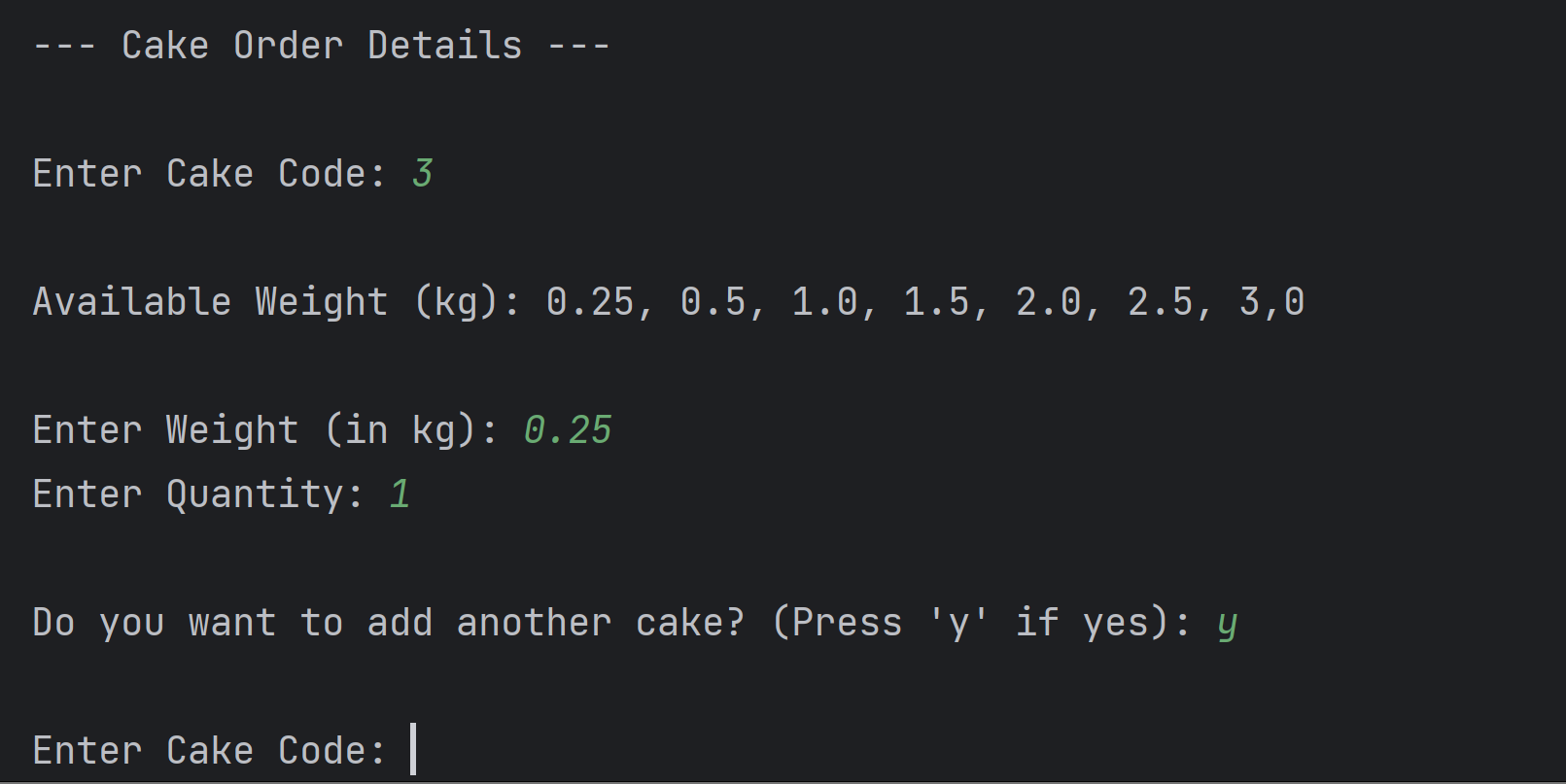
Example 2: any alphabets (strings) is entered

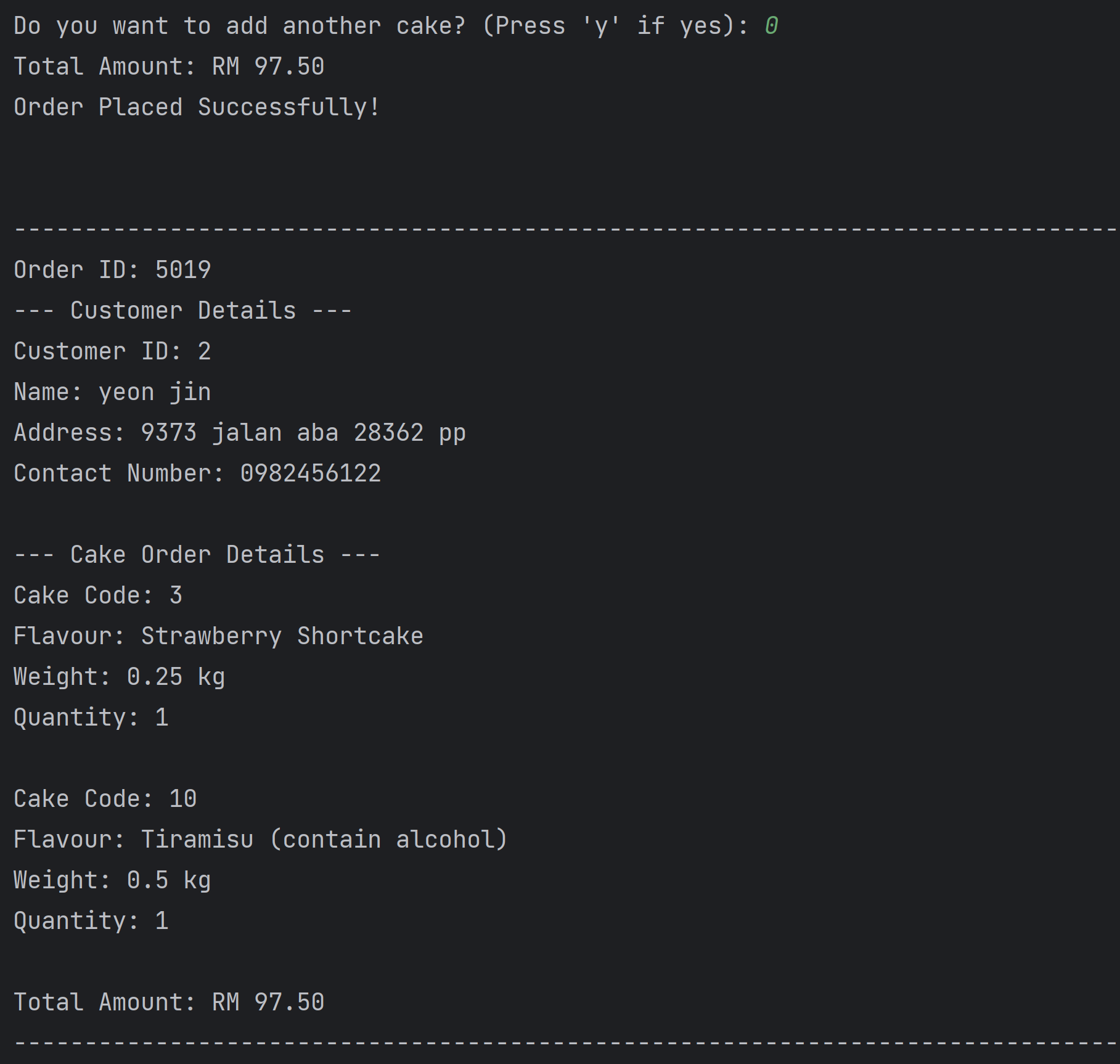


Example 3: any numbers is entered



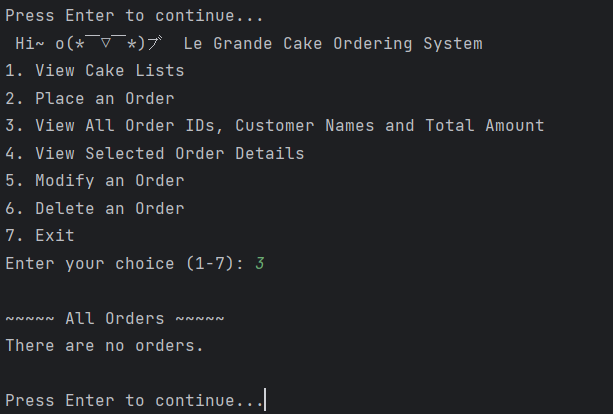
When the user wishes to add more cakes to the order, the programme will loop on the same questions as when the first cake order is added. After the second cake is added, the programme will ask again if the user wants to add another cake or not. If not, it will print the order details.



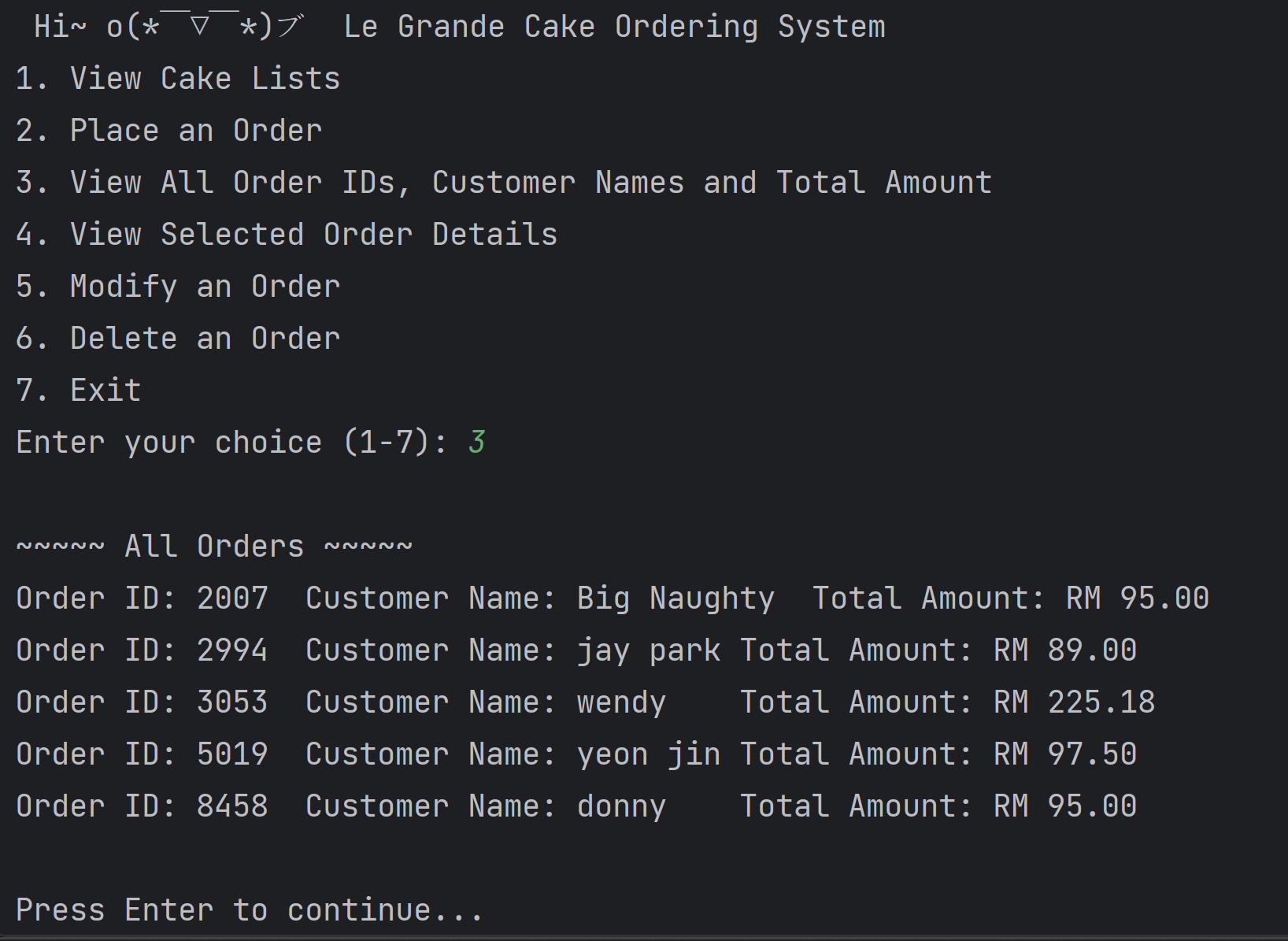


1. When user input “3” (view all orders)

If there are no orders in the BST, it will show the message that “There are no orders.”

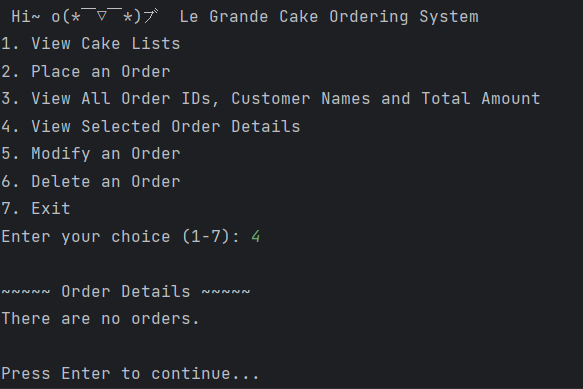


If there are orders in the BST, it will display all the orders, including the order ID, customer name, and total amount, using in-order traversal.

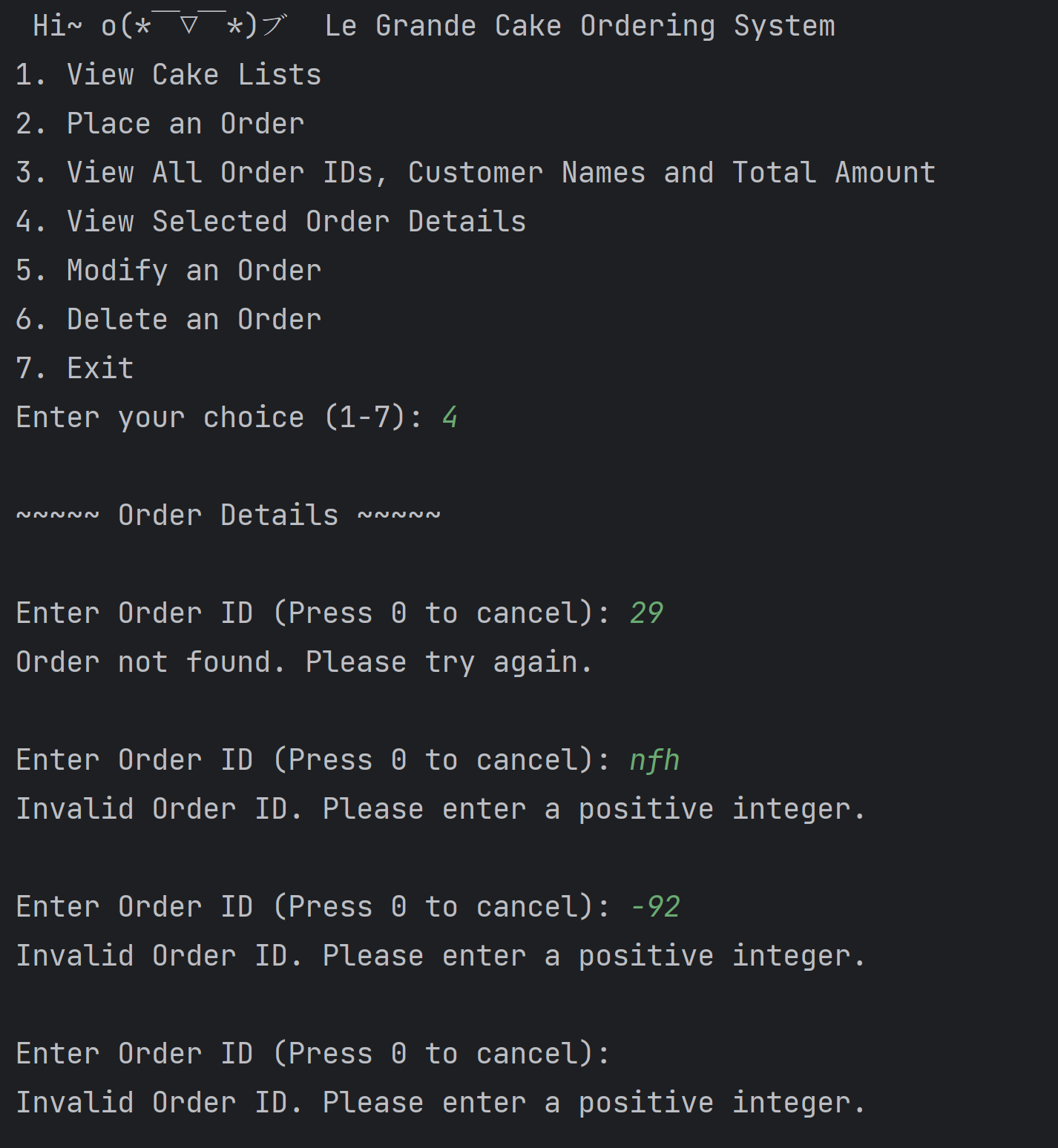


1. When user input “4” (view a specific order)

If there are no orders in the BST, it will show the message that “There are no orders.”

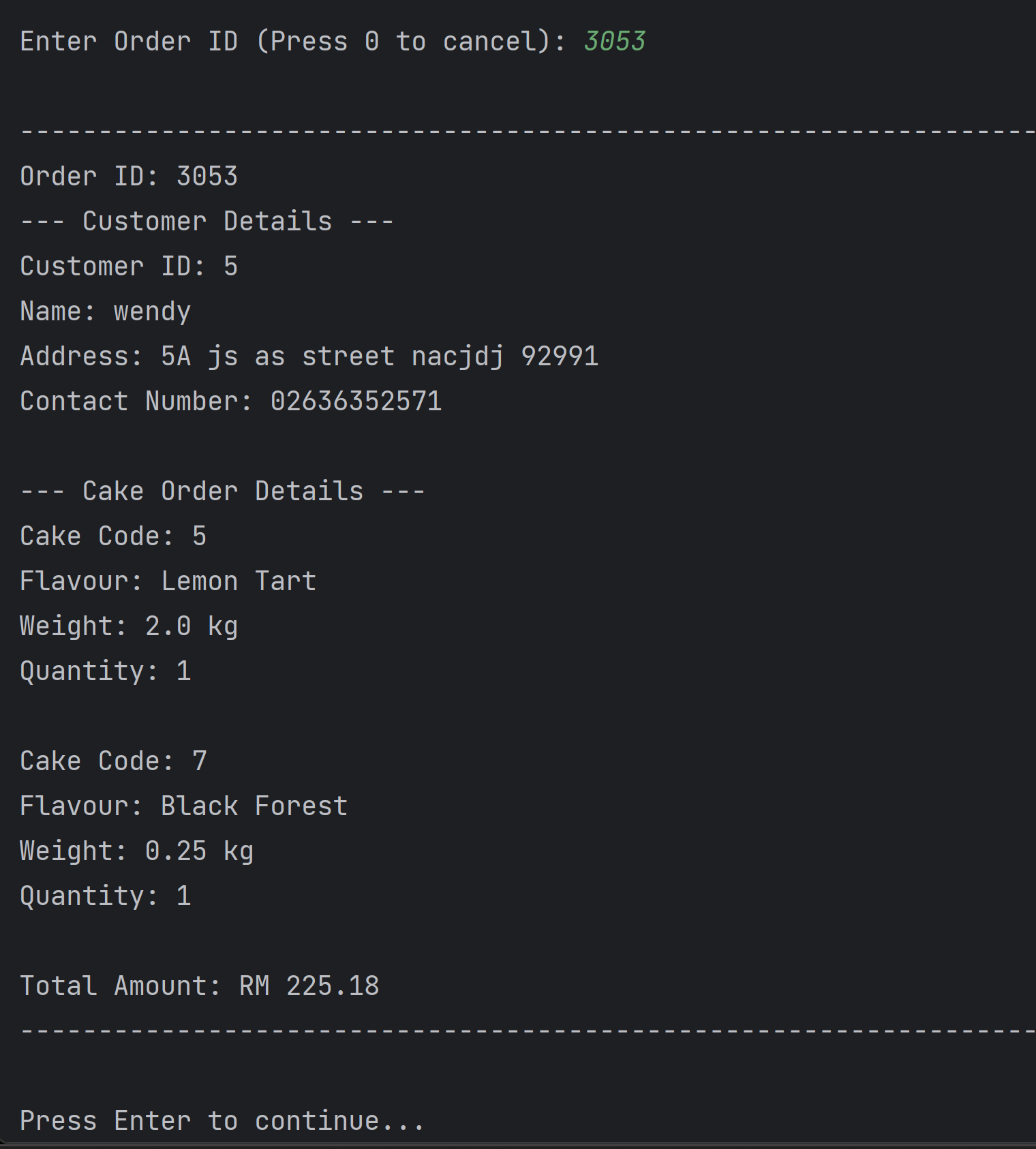


Based on the screenshot output below, when there are orders inside the BST, it will ask for the user input the order ID, and then the programme will validate the input. If the input is empty, not an integer, or a negative integer, the programme will show the message "Invalid Order ID. Please enter a positive integer.". However, if the input is an integer, the programme will search on the integer, and if the order is not in the BST, it will show "Order not found. Please try again.".

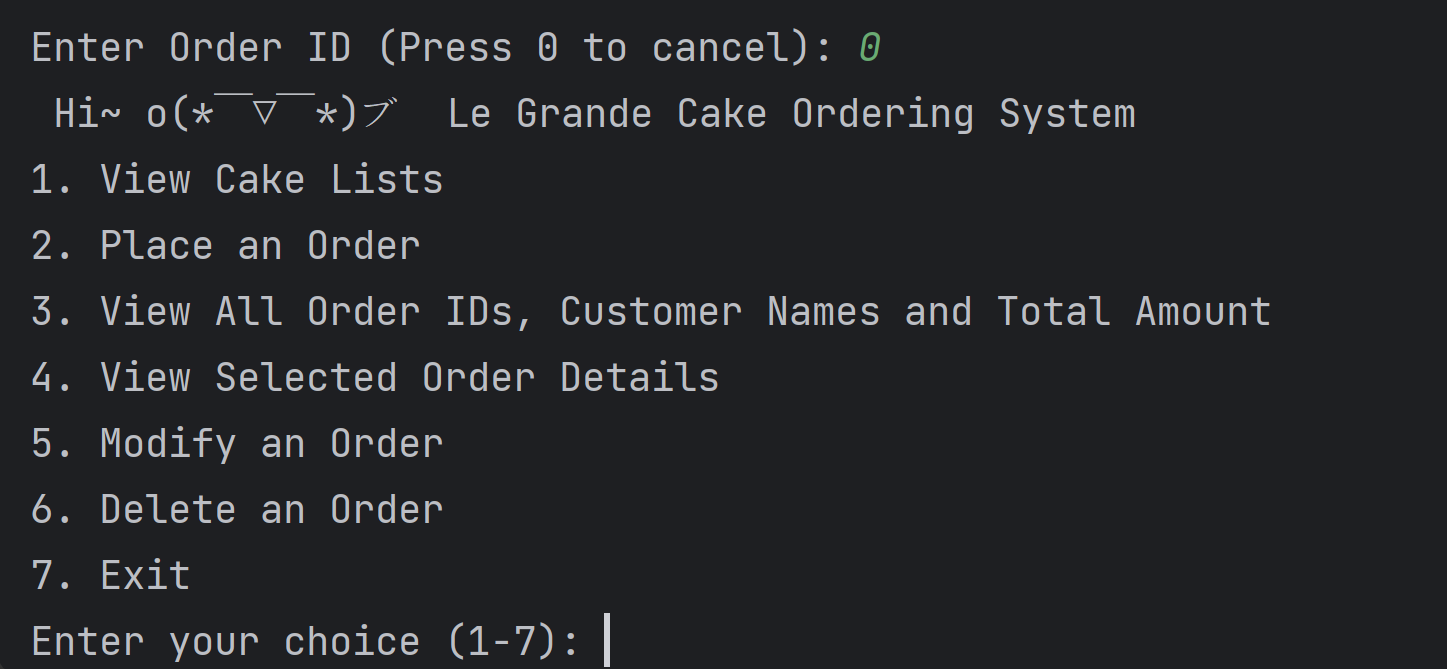


If a valid order ID is entered in the BST, the order details will print out, showing the order ID, customer details, cake order details, and the total amount. The screenshots of output below show when an order has only one cake order and multiple cake orders. After viewing the order details, the user is required to press ‘enter’ to return to the starting menu.



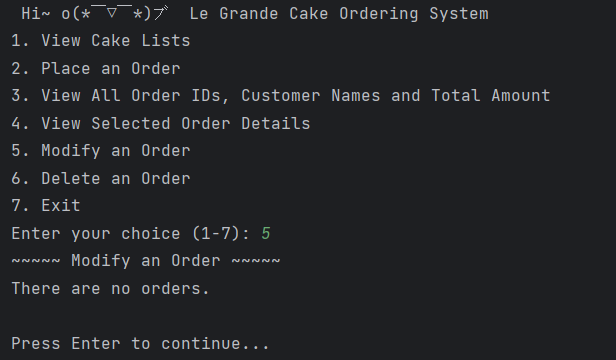


If user want to terminate this function, the user is required to enter “0”. Then it will print the starting menu again and asked for user input to choose another function.

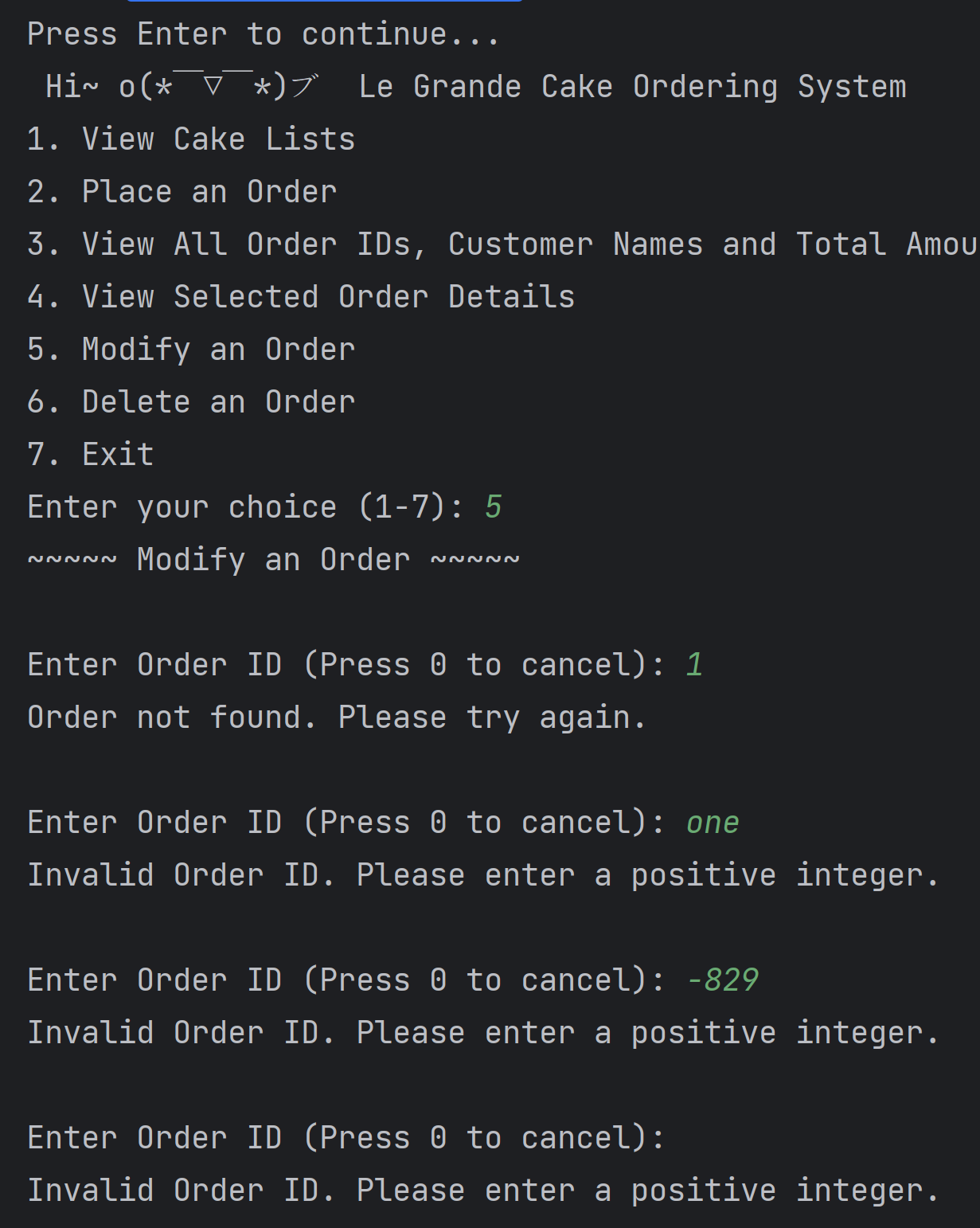


1. When user input “5” (modify an order)

If there are no orders in the BST, it will show the message that “There are no orders.”.



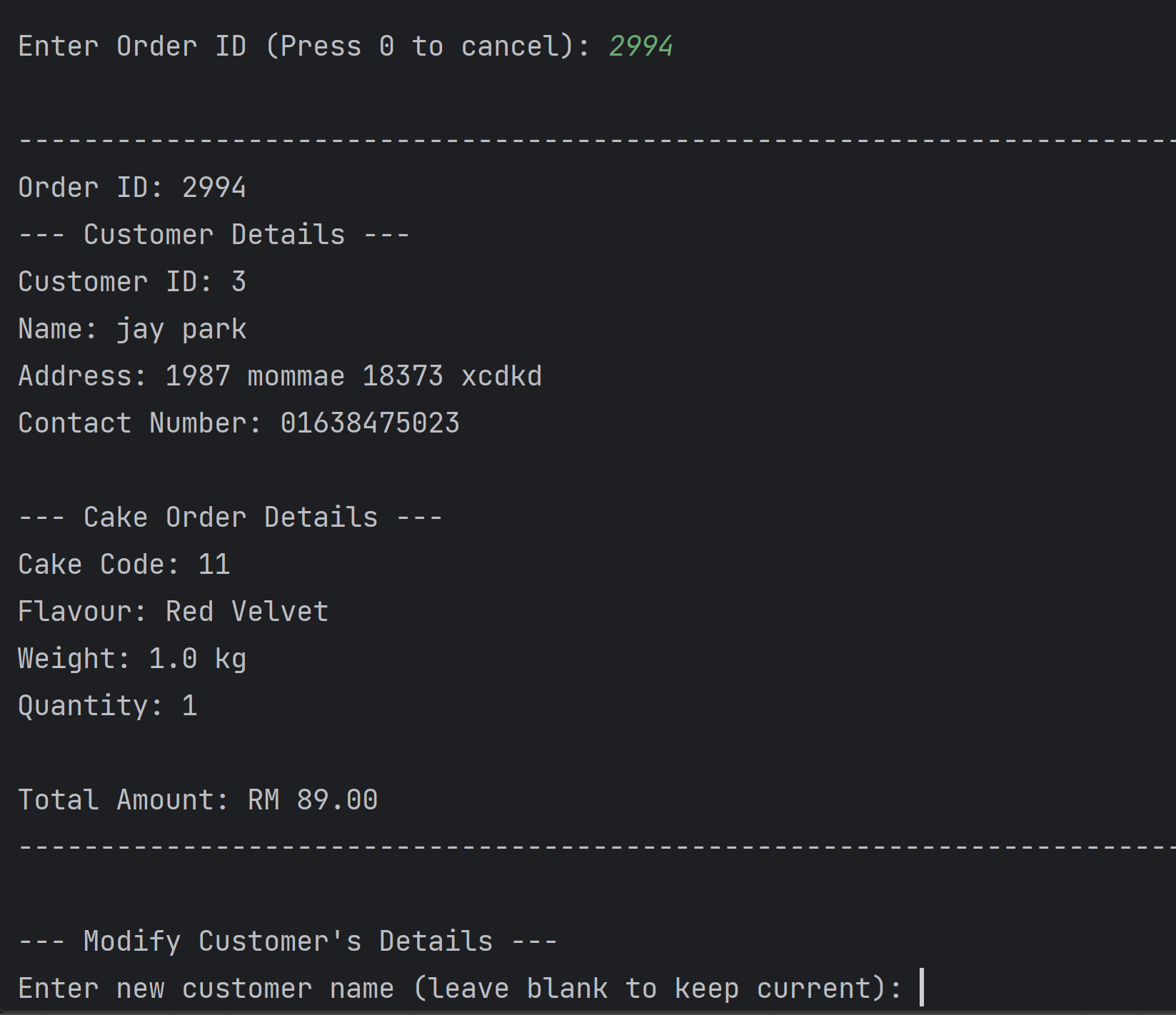
Based on the screenshot output below, when there are orders inside the BST, it will ask for user input the order id, then the program will validate on the input. If the input is empty, not integer, or negative integer, the programme will show the message “Invalid Order ID. Please enter a positive integer.”. However, if the input is integer, the program will search on the integer, while the order is not in the BST, it will show that “Order not found. Please try again.”. If the user needs to end this function press ‘enter’ and it will back to the staring menu.

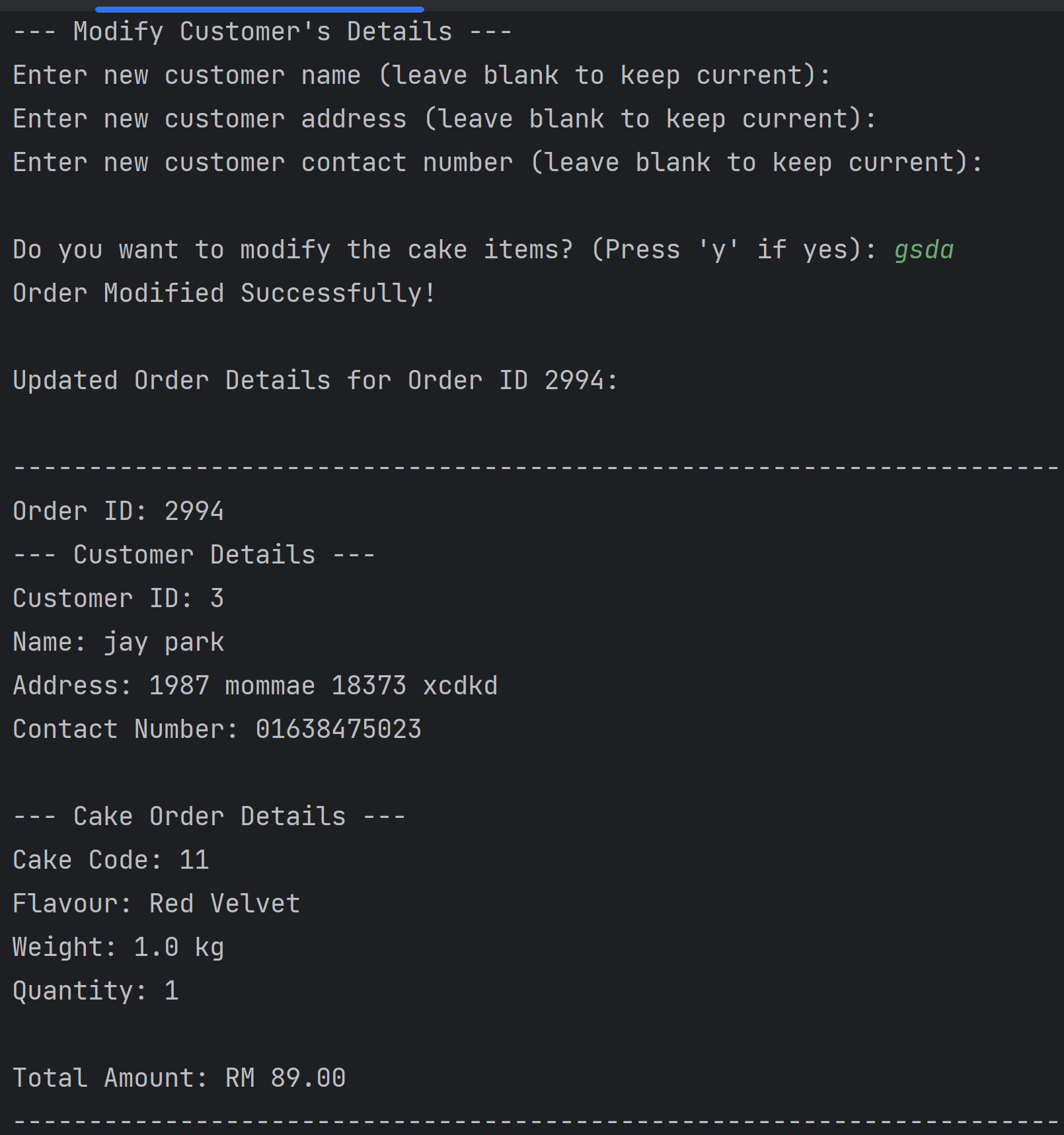




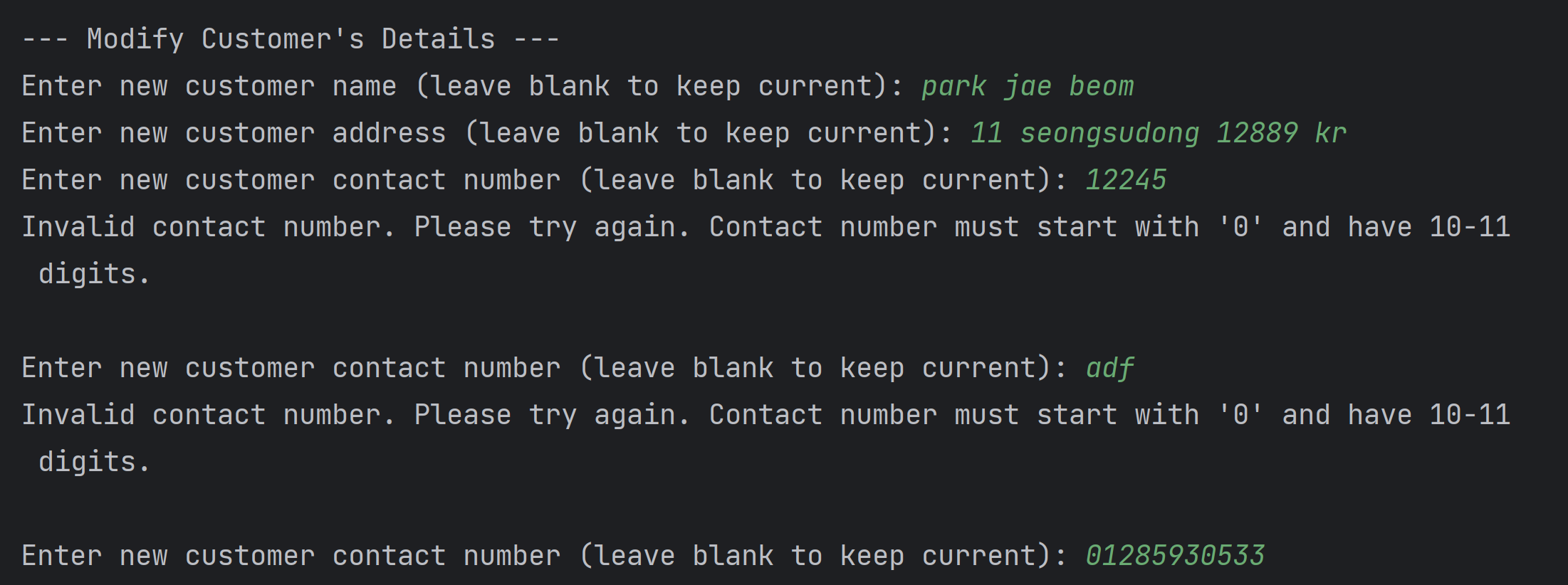
If a valid order ID is entered in the BST, the order details will print out, showing the order ID, customer details, cake order details, and the total amount. Next, the programme will ask the user to enter the new value for modifying the customer’s details; if the user wants to remain with the same value, press ‘enter’ to leave it blank. After the customer’s details are modified, the programme will ask if the user wants to edit any cake orders. If the input is other than ‘y’ or ‘Y’, the programme will consider that the user is not going to modify the cake order. Thus, the order will update to the BST and the updated order details will display.

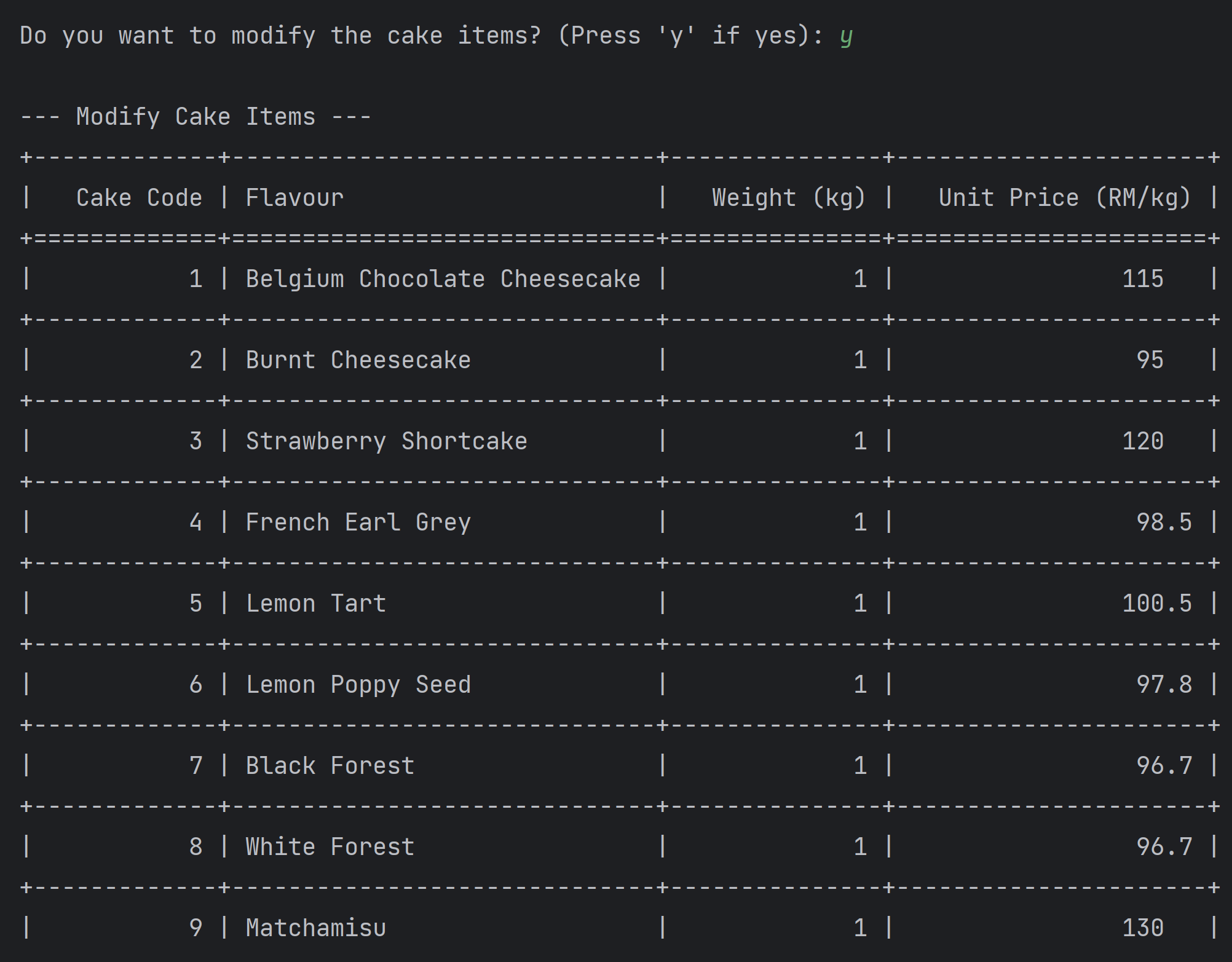
Example 1: Remain unchanged

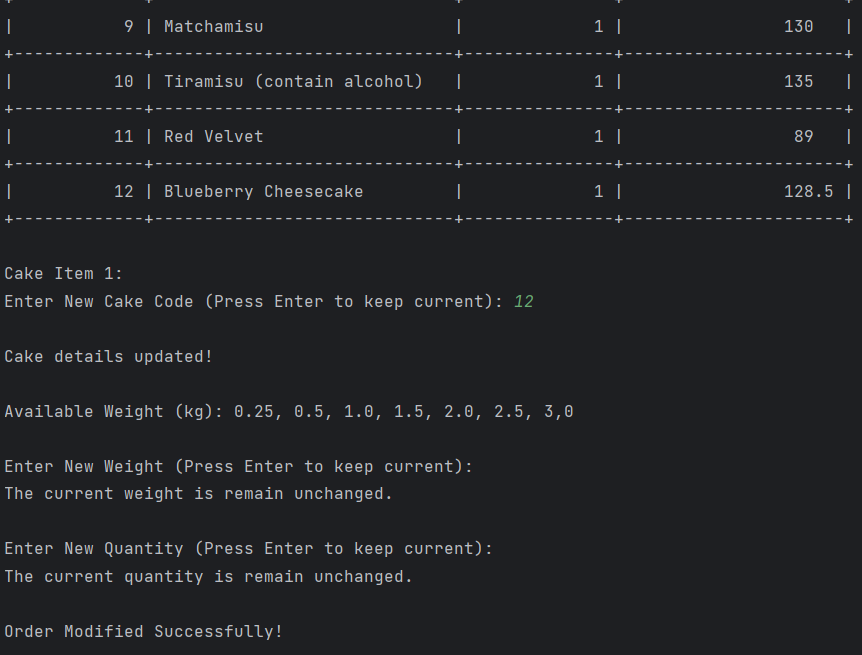


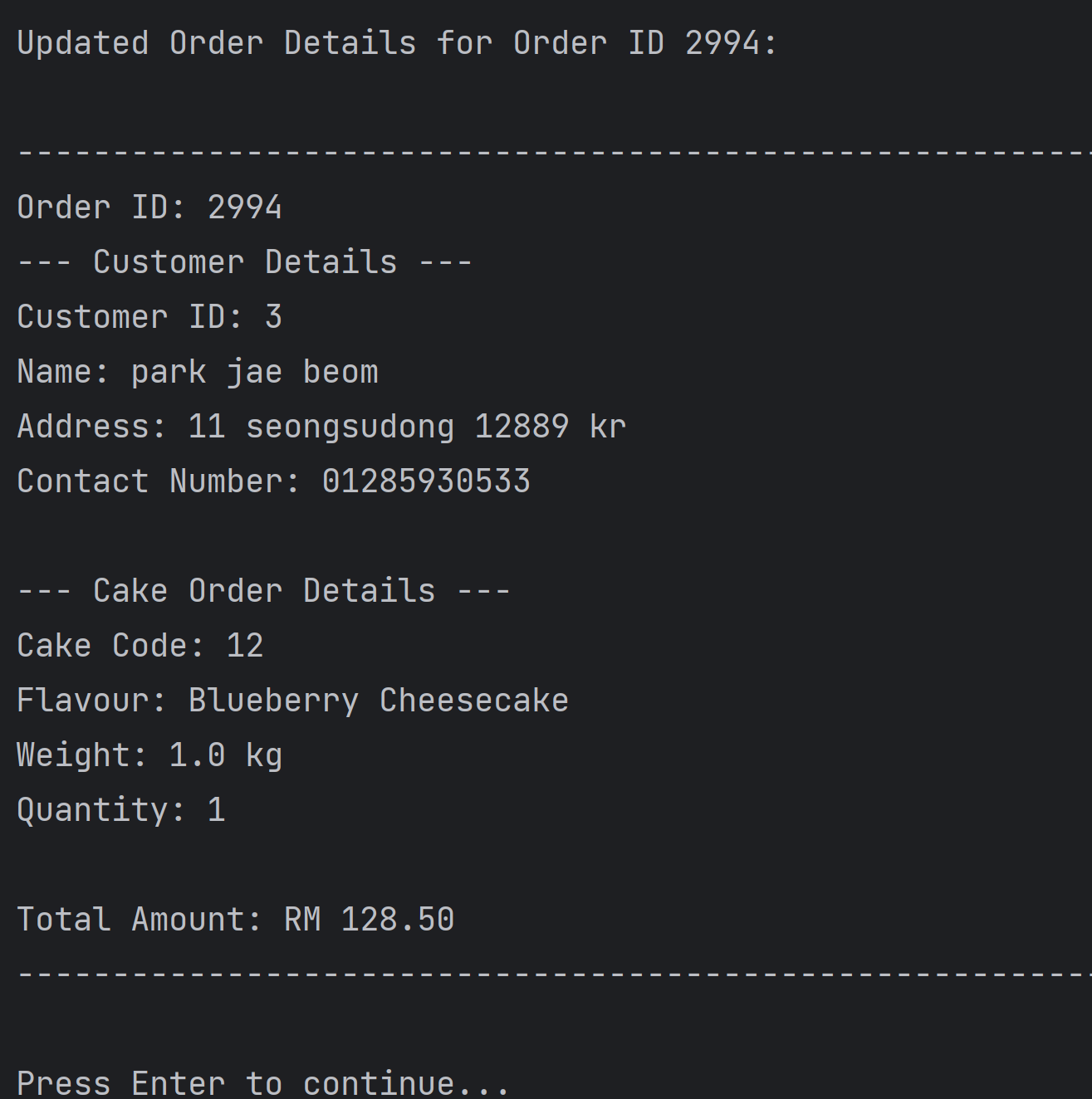


Example 2: Changing on the customer’s details and the cake code (flavour) where the weight and quantity remain unchanged.



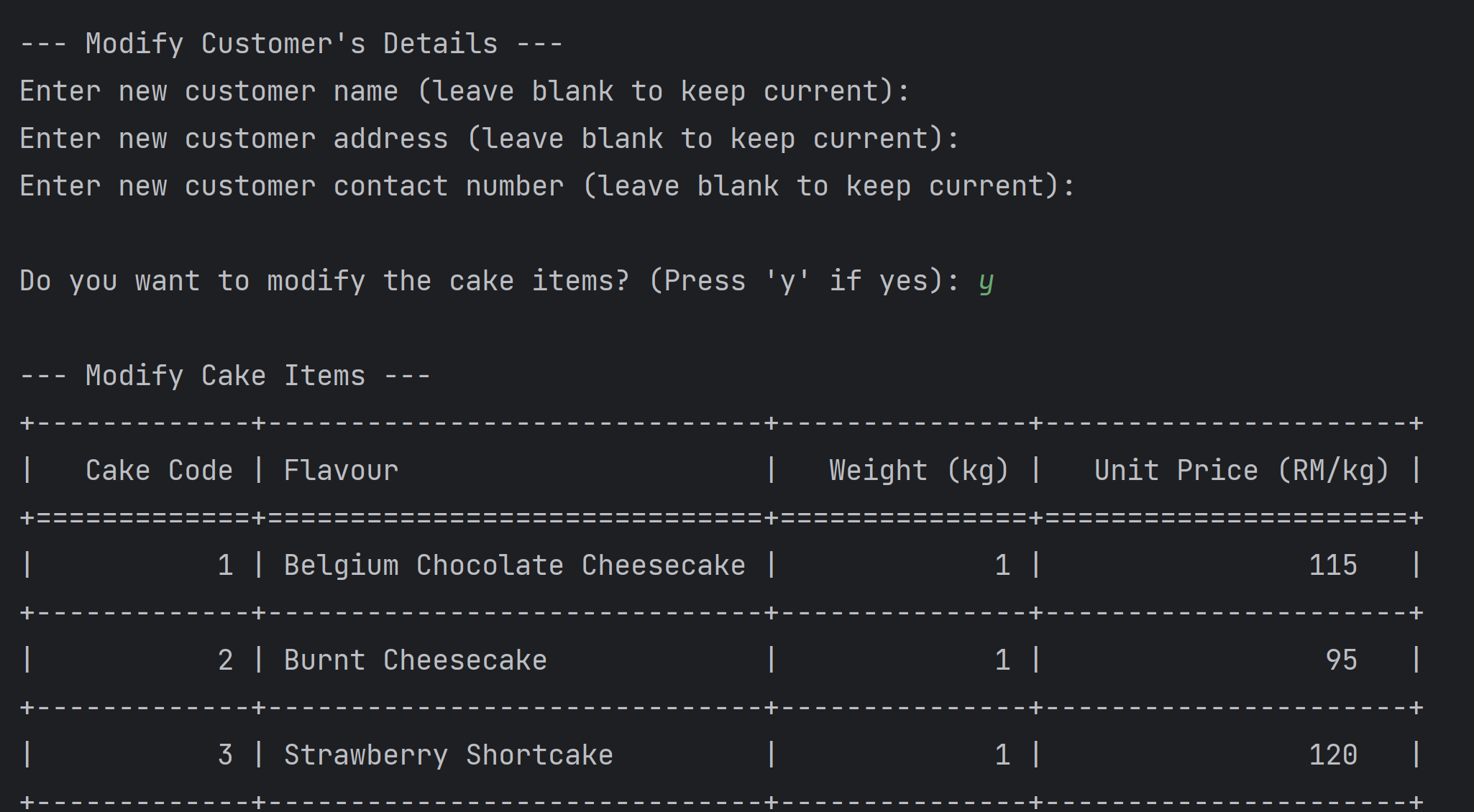


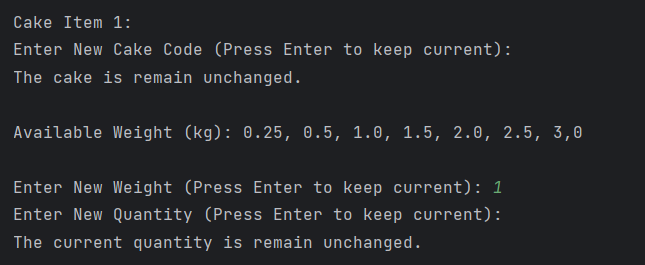


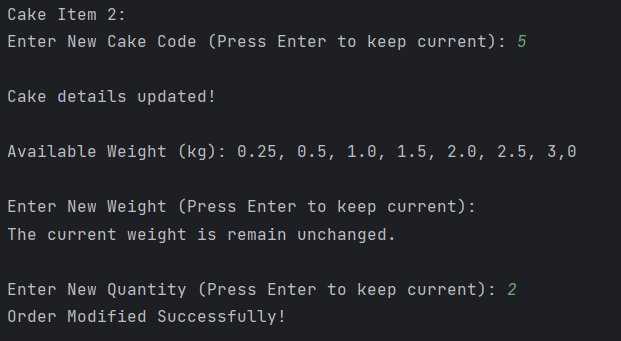


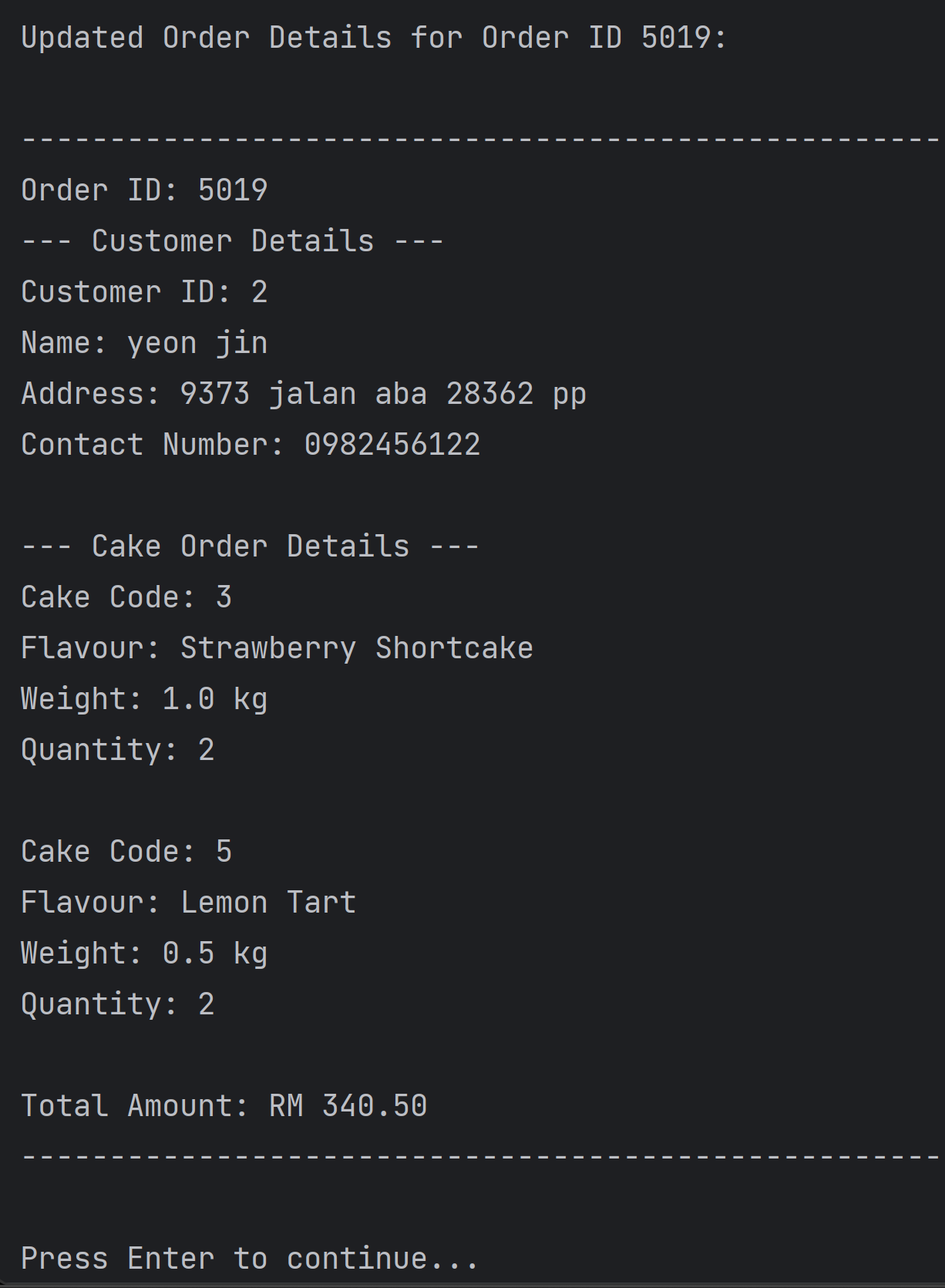
Example 3: Changing on the order that have multiple cakes.





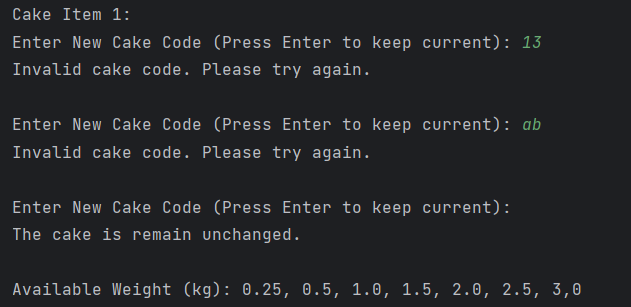


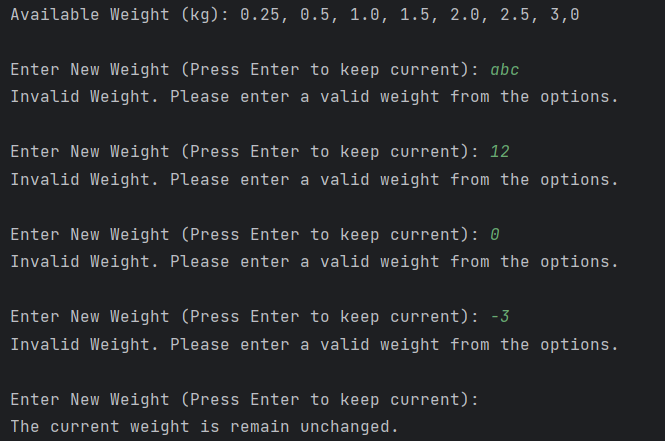


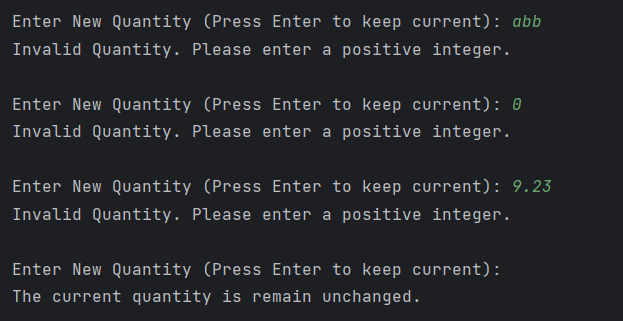


Error handling on the cake items (cake code, weight, quantity):

If a new cake code enter is not in the range. User is prompt to reinput the cake code again. If the user changes the mind to not to change the cake flavour, then it can still press enter to remain the same cake choice. Same for the weight and quantity, the system will check on the input where the weight must in the range of 0.25kg, 0.5kg, 1.0kg, 1.5kg, 2.0kg, 2.5kg, 3.0kg, and the quantity must a positive integer that more than 0.

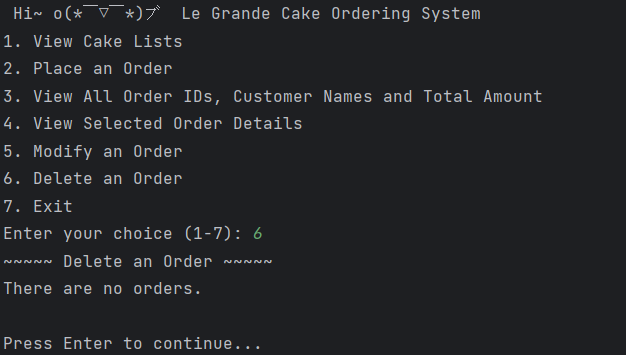




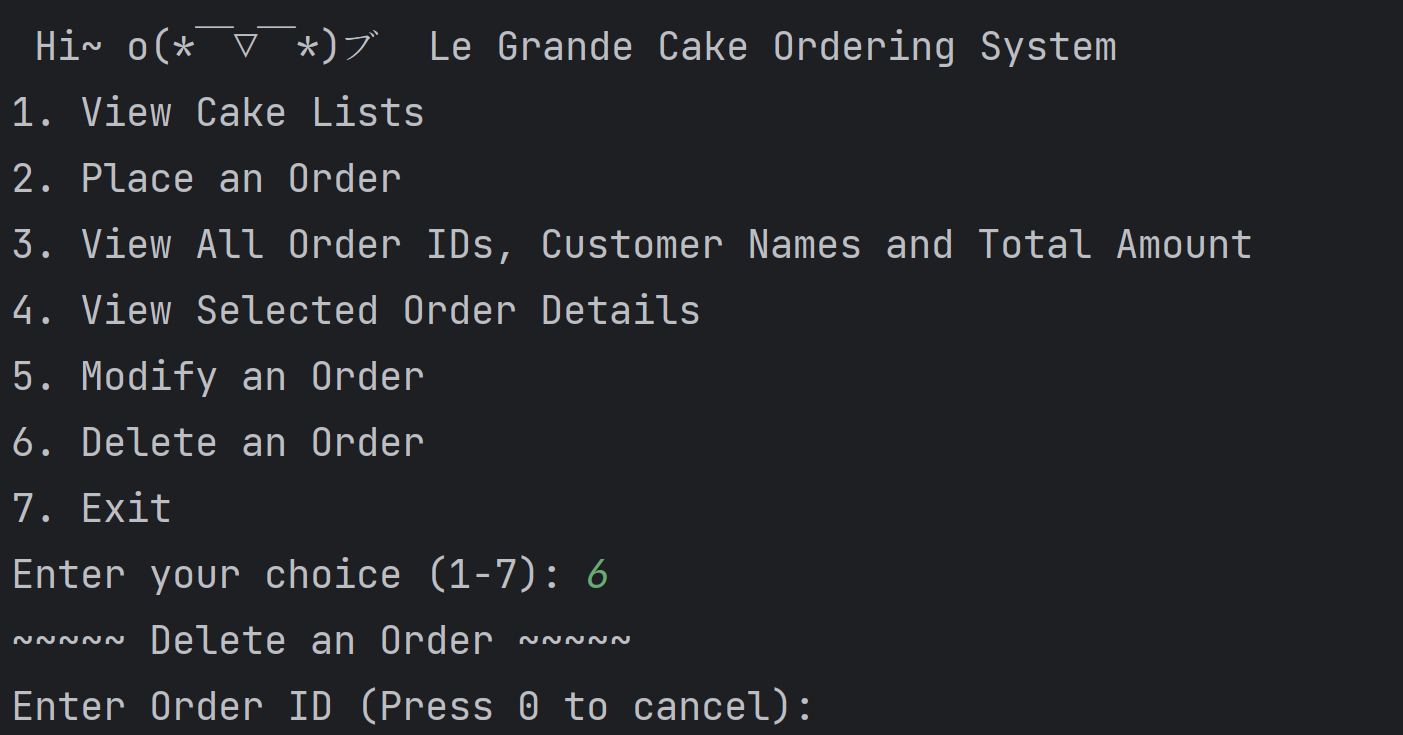


1. When user input “6” (delete an order)

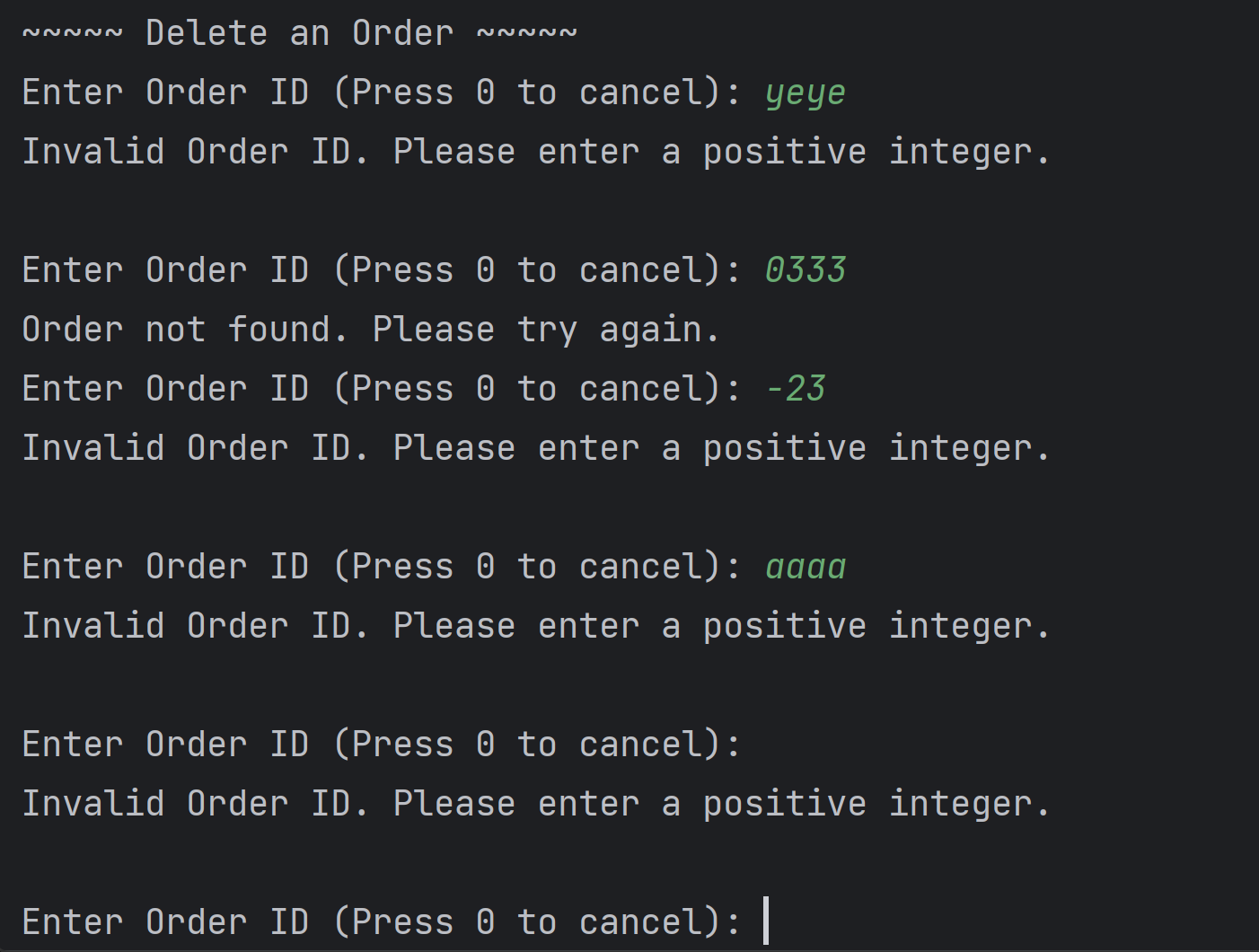
If there are no orders in the BST, it will show the message that “There are no orders.”



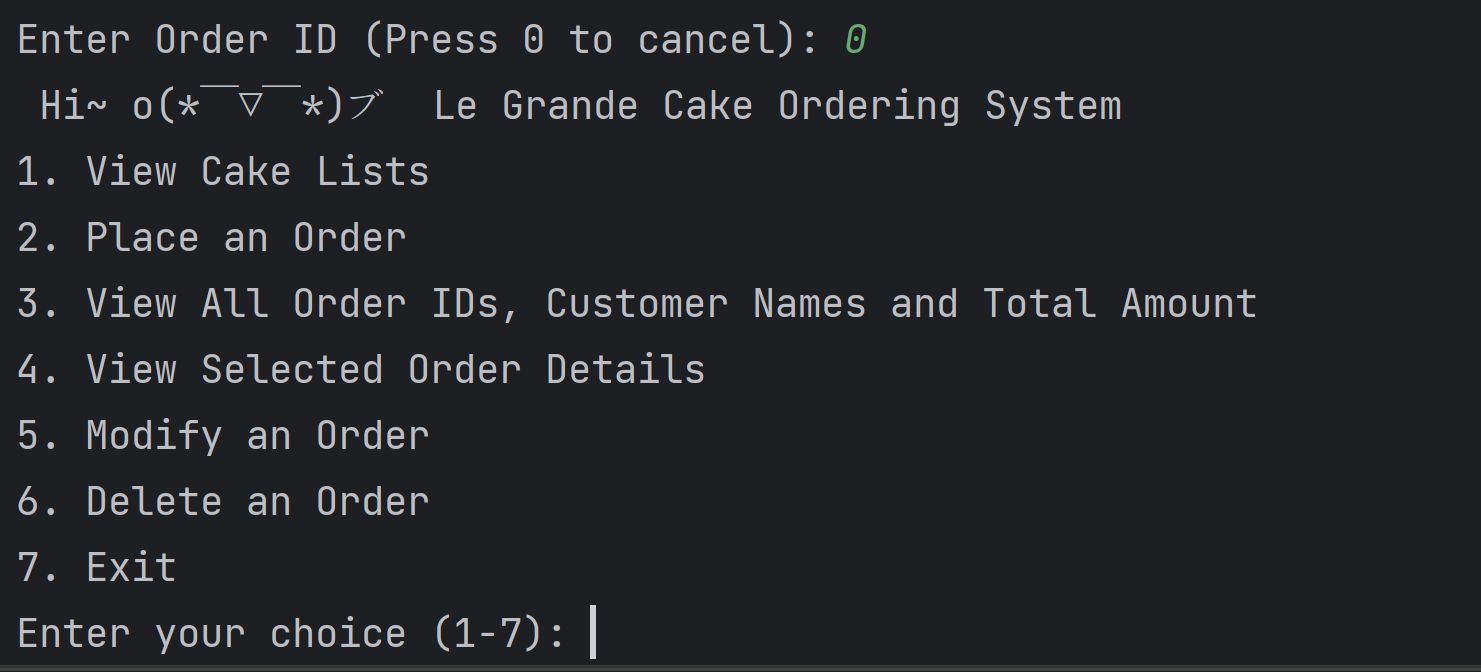
Based on the screenshot output below, when there are orders inside the BST, it will ask for the user's input of the order ID, and then the programme will validate the input.



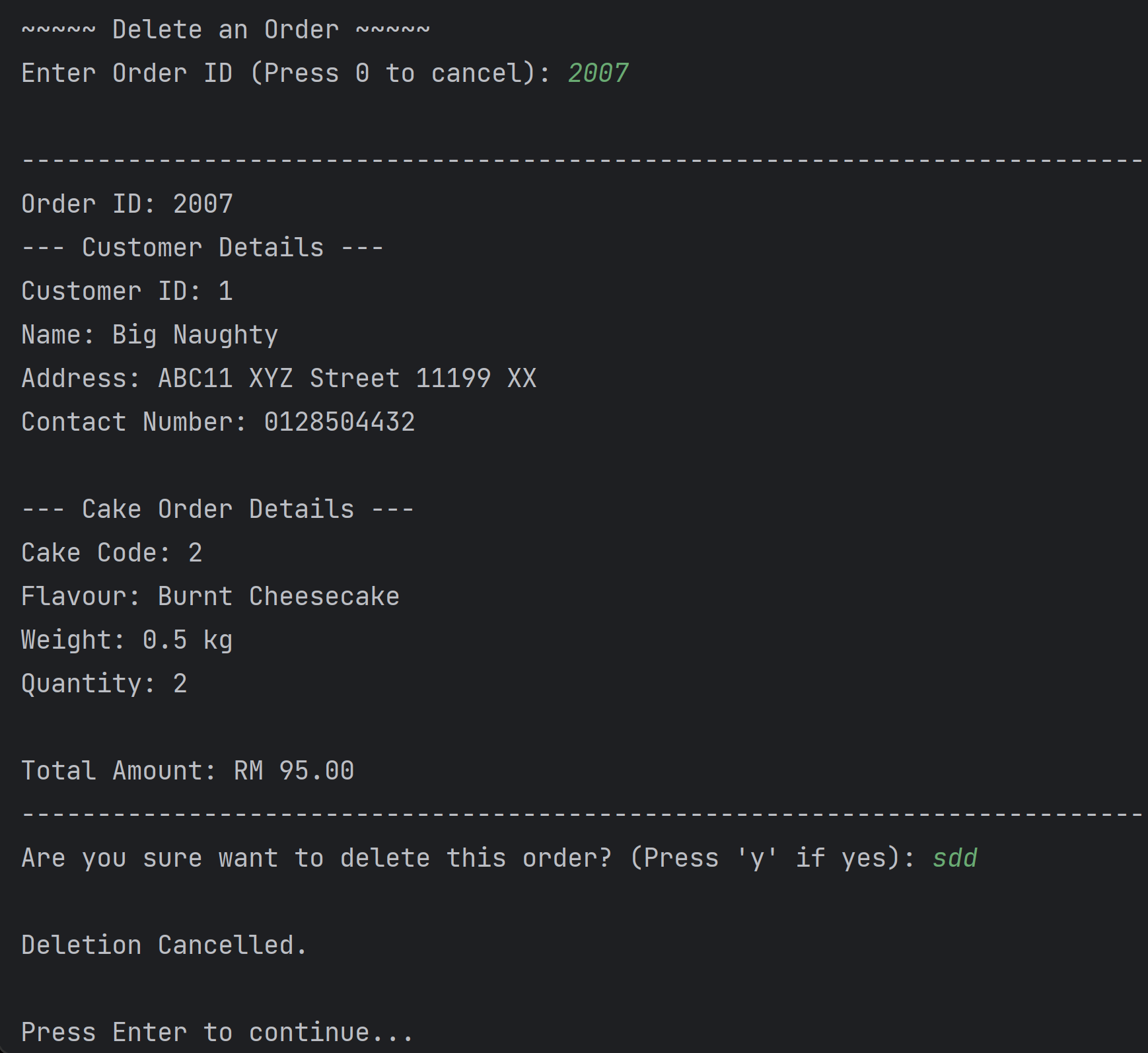
If the input is empty, not integer, or negative integer, the programme will show the message “Invalid Order ID. Please enter a positive integer.”. However, if the input is integer, the program will search on the integer, while the order is not in the BST, it will show that “Order not found. Please try again.”.

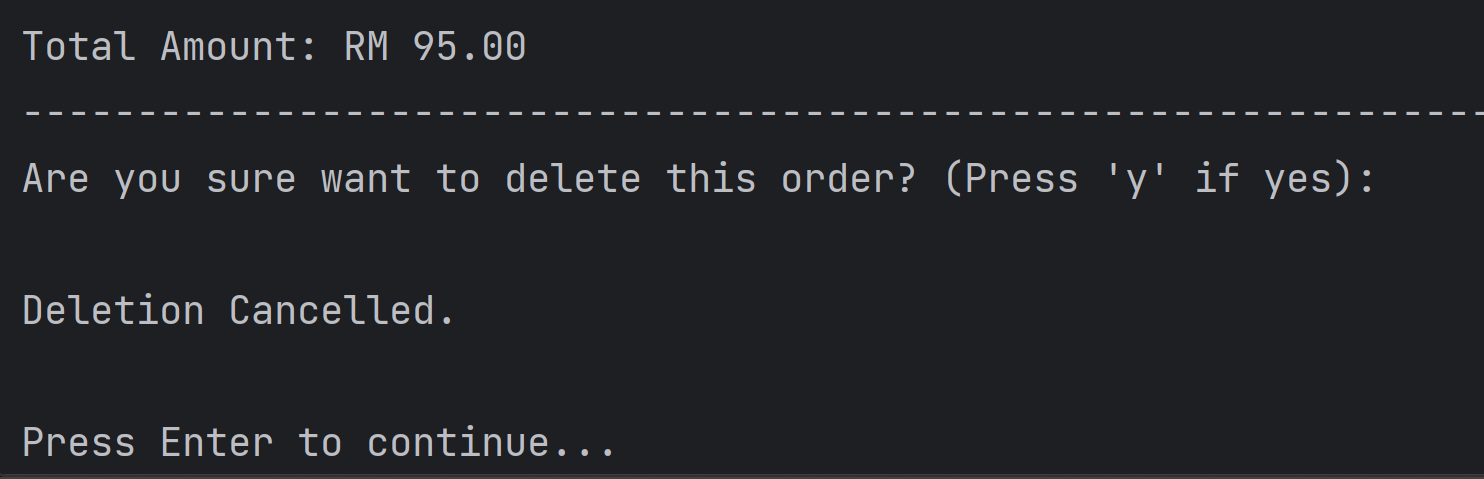


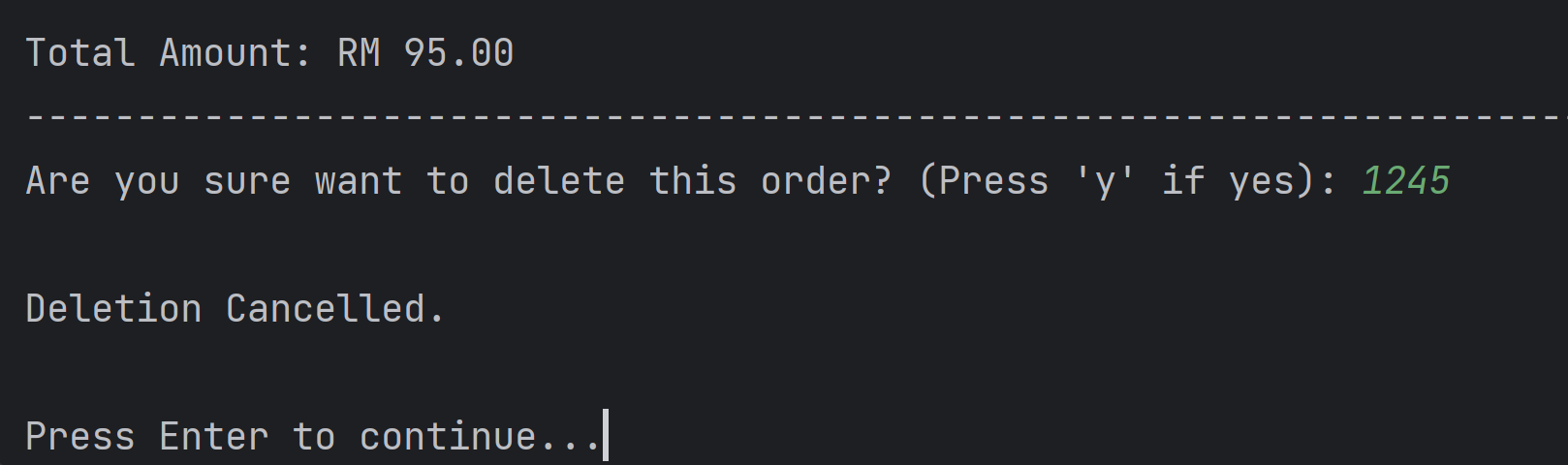
If the user wishes to end this function, the user is required to enter "0". Then it will print the starting menu again and ask for user input to choose another function.



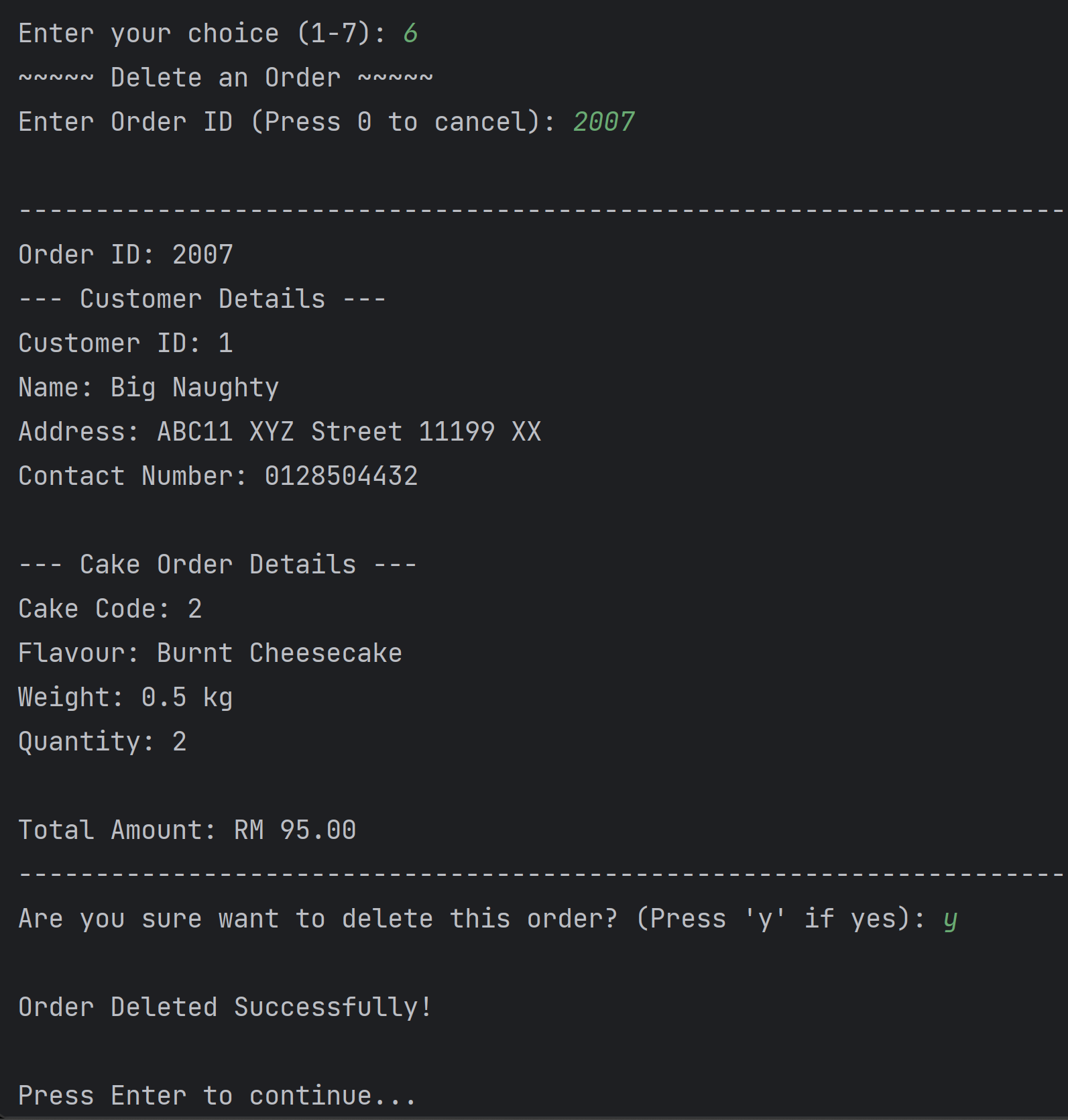
The order details for the order ID entered will be display if the order exists in the BST. Next, the system will ask for the user is really wanted to delete the order. If the input is other than ‘y’ or ‘Y’, the programme will consider that the user is not going to delete the order.

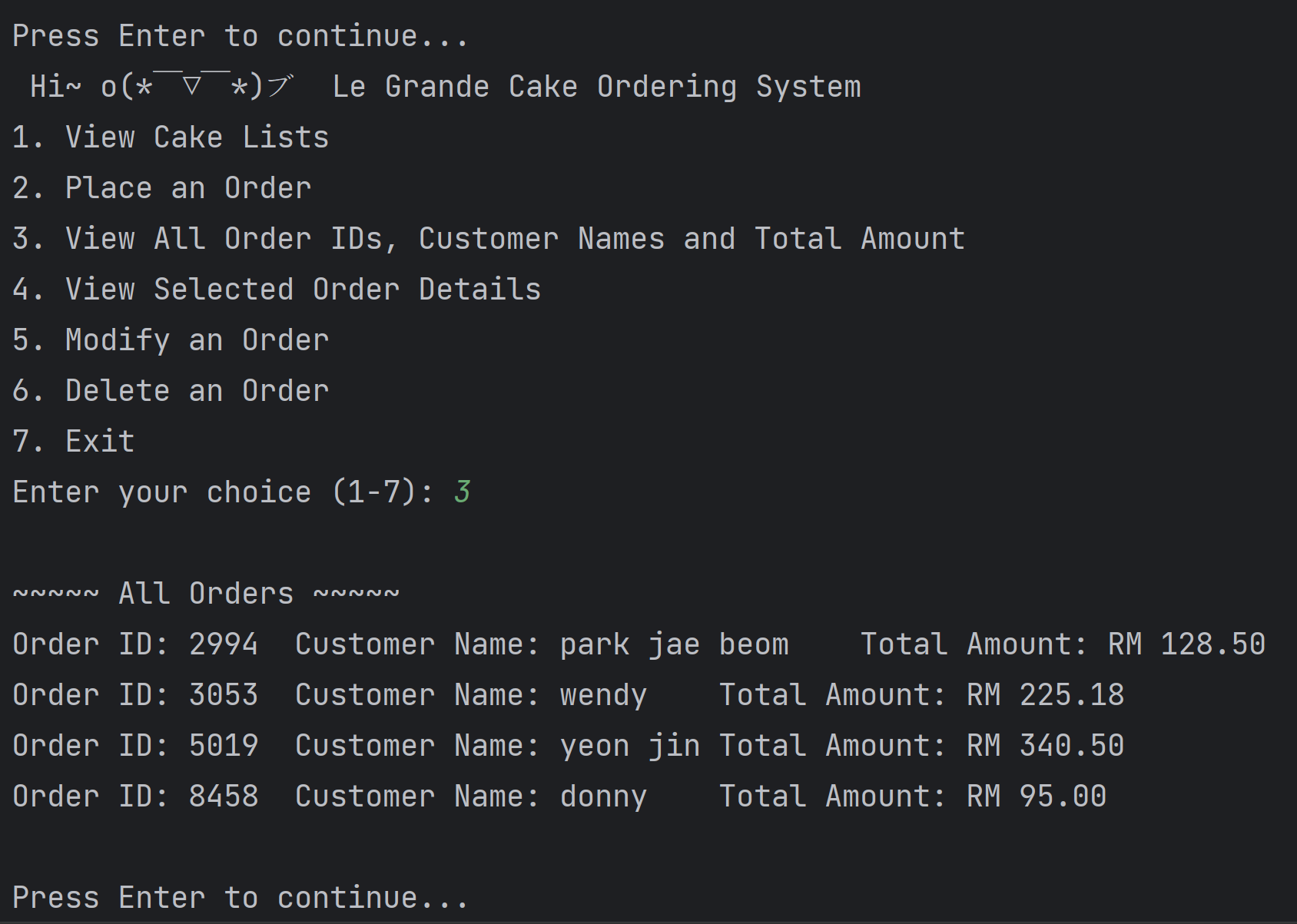


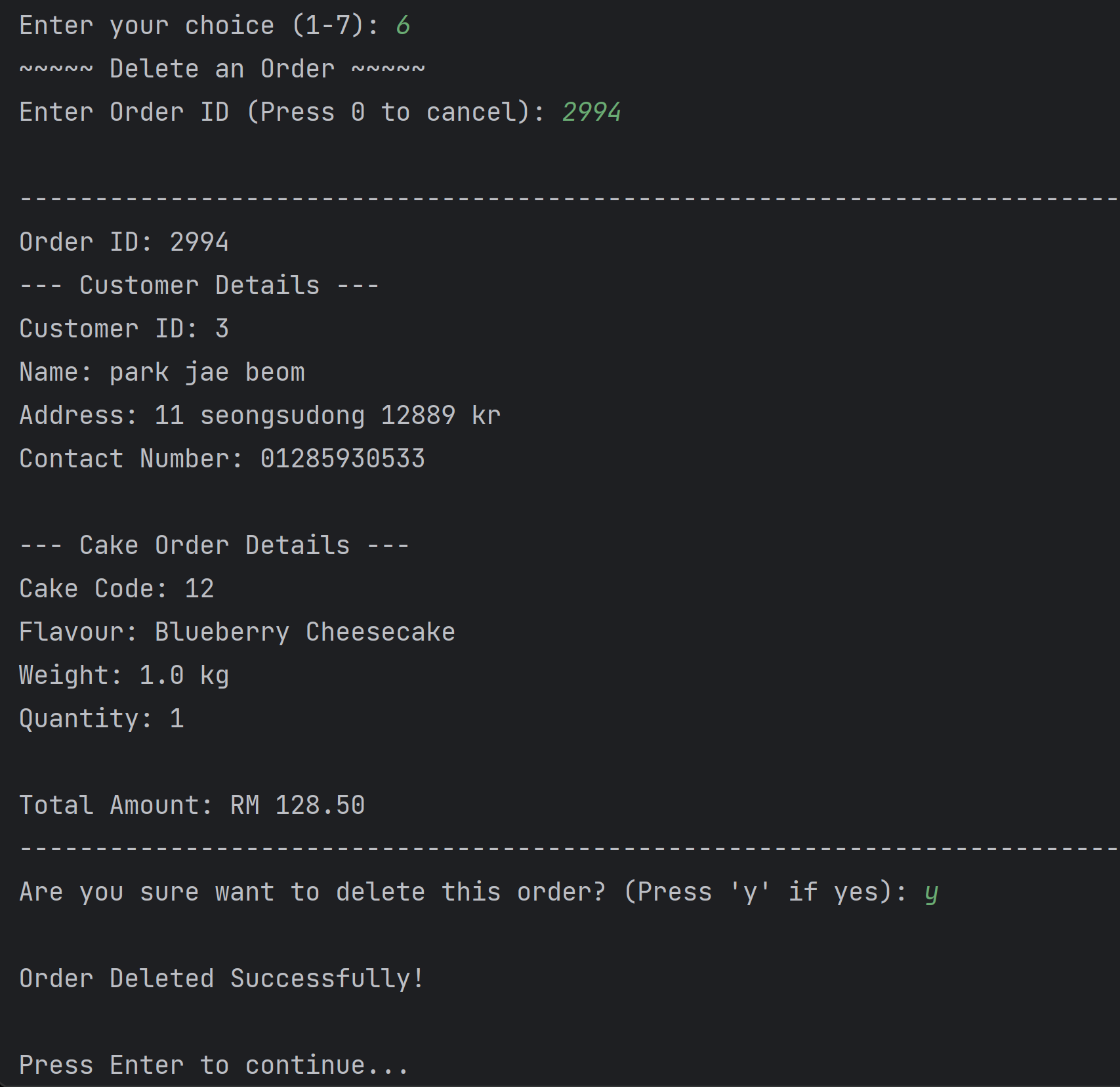




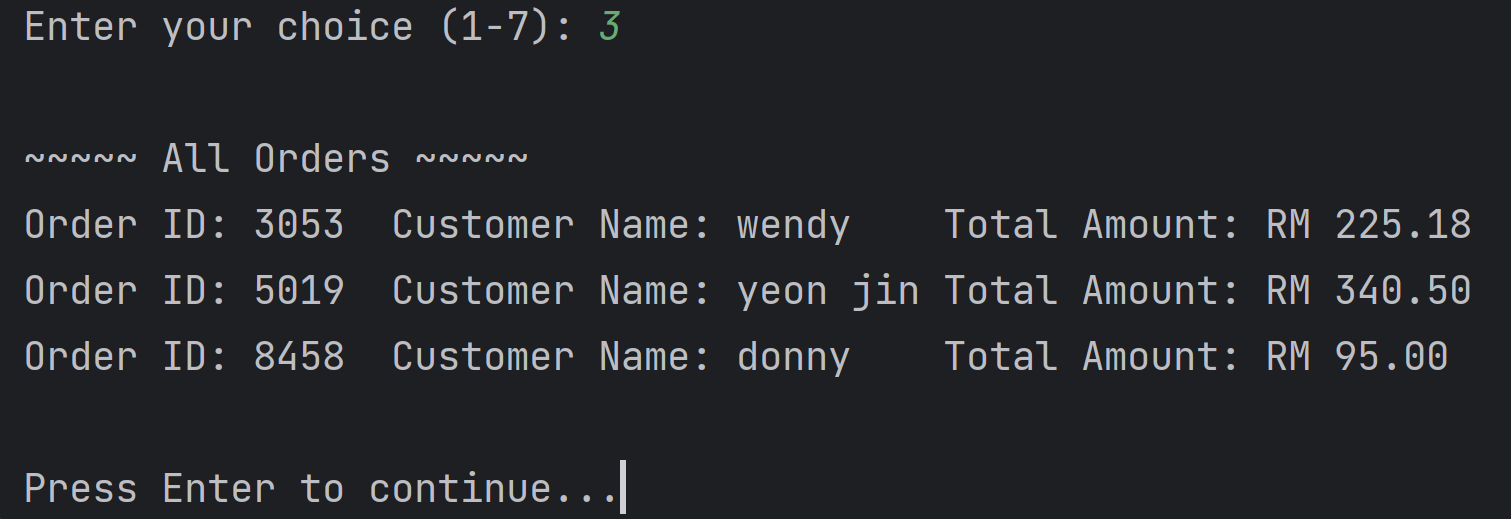
If the answer is ‘y’ or ‘Y’, the order will be deleted from the BST. Since there are few conditions that can take place to delete the order in BST. All the outputs below show the deletion of orders that take placed in the BST.

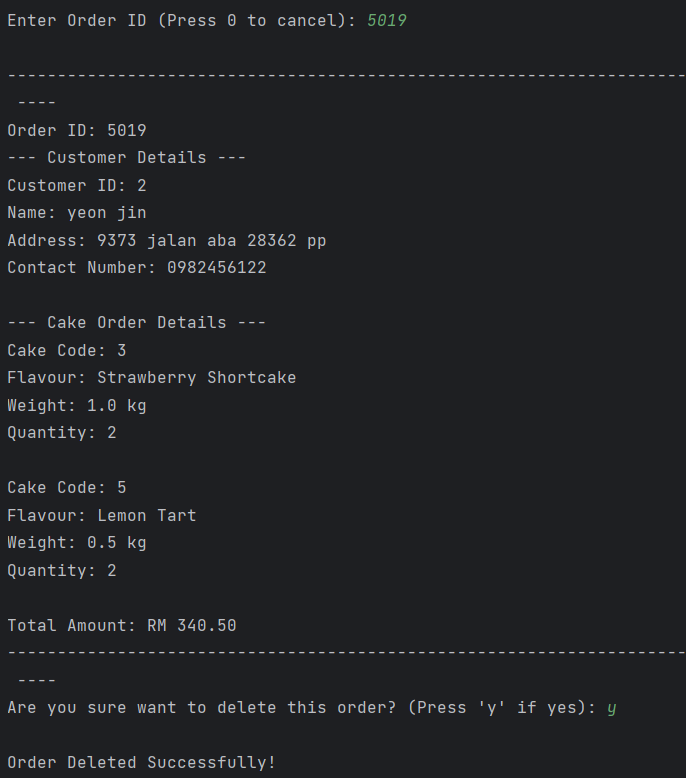
  
The order lists after deletion:



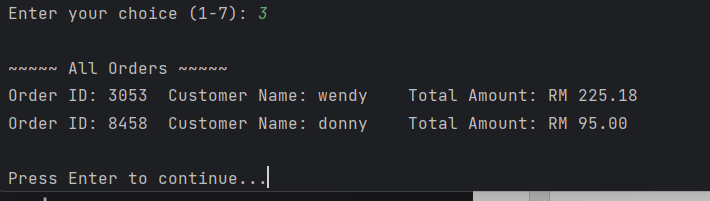


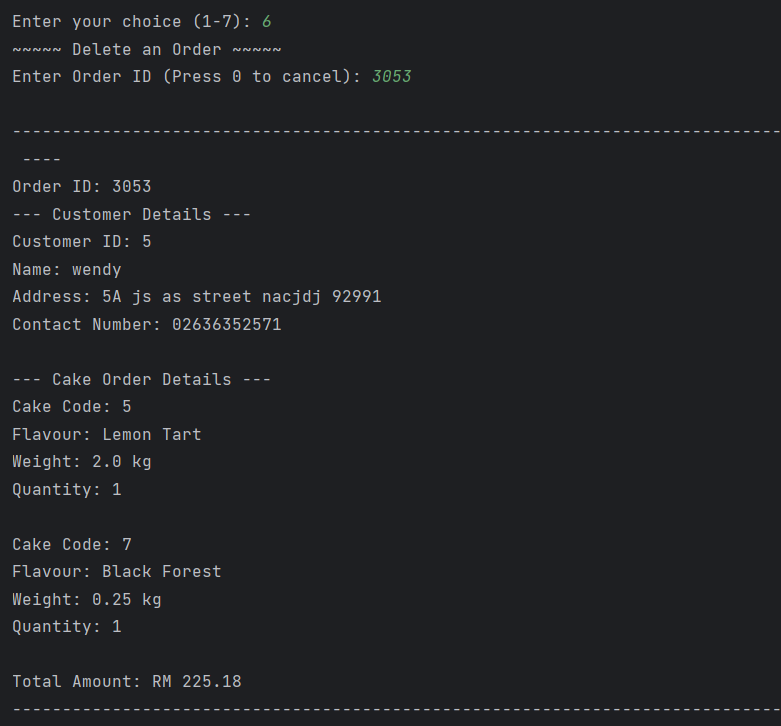
The order lists after deletion:

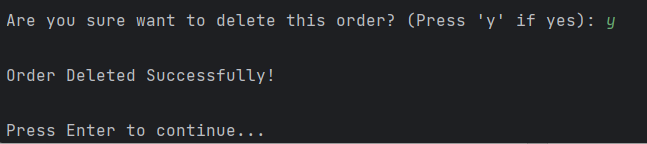




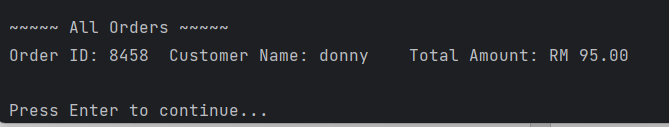
The order lists after deletion:







The order lists after deletion:

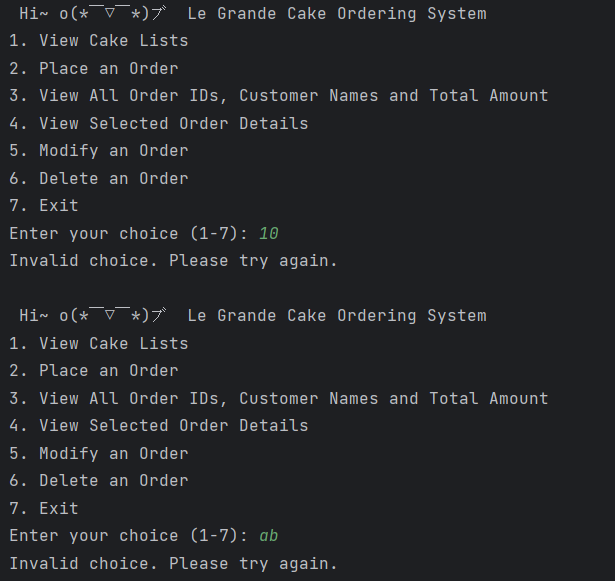


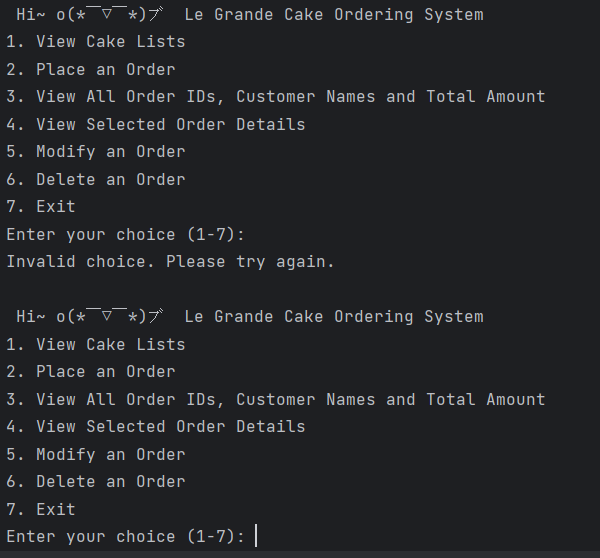
1. When user input “7” (exit the program)



1. If user input numbers out of the ranges or not integer.

If the invalid values are inserted, the program will loop to prompt for user input until the valid choice (1 to 7) is entered.





### Weakness of solution

The program does not include the clear screen function after each action is completed. Therefore, the screen will be very messy if the program has run multiple times. This will made the system difficult to navigate and read. The programme should include a screen cleaning function to improve user experience.

Next, there are no validation checks in the programme for the customer's name and address inputs. The programme only validated that the customer’s name and address should not be empty. This implies that the programme accepts any strings input by the user without validating its accuracy or completeness. Validation tests must be implemented to guarantee that the customer's name and address inputs fulfil the relevant parameters.

Besides that, the Customer class has no searching method in the code. As a result, there is no method to determine whether a customer is already exists in the system. Therefore, redundant customer profiles with different customer IDs are possible, which resulting in data inconsistency and confusion. For the improvement, it also can include the BST for the Customer class to insert, search, and delete the customers.

Lastly, the cake lists are already fixed in the system, it does not have the flexibility to add or remove cakes from the lists. This restricts the ability of the system to adjust to shifts in the cake selections provided. To further enhance the solution, include the function that allows for the dynamic addition and removal of cakes, enabling the system more versatility and scalability.

### Code References

(Lane, 2021)

(Rastogik346, n.d.)

(Kumar, 2021)

(Bhat, n.d.)

(PranchalK, n.d.)

## Question 2: Hash Table Collision technique average cost comparison

Write a program to compute and compare two collision techniques: Chaining and open addressing (linear probing) for the hash table implementations.

The program shall insert 5,000 randomly generated numbers into two separate hash table objects that use different methods (chaining and open addressing) to handle the collision. Use 6001 as the table size of the hash tables.

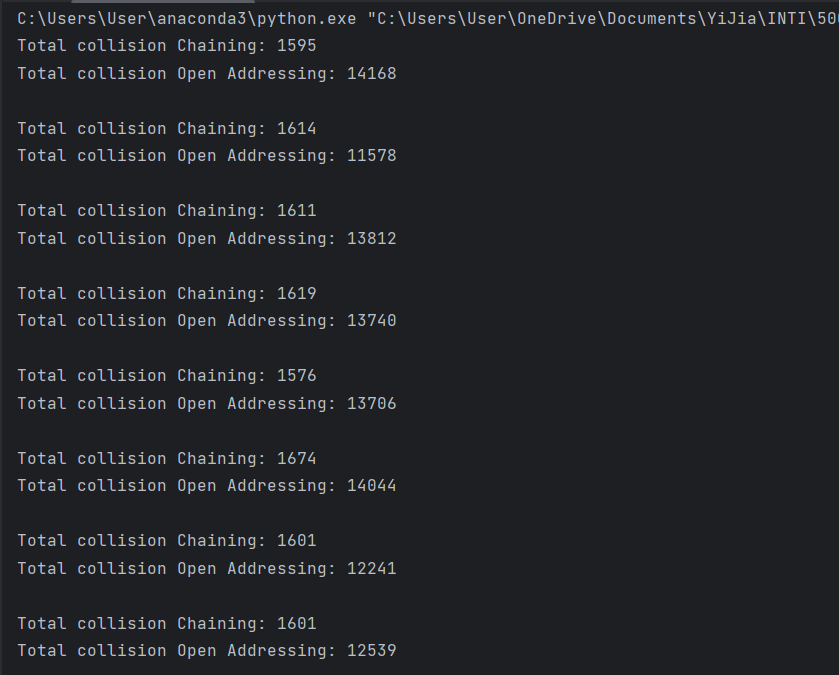
The program shall execute at least 10 times and calculate the maximum, minimum and average number of collisions that had happened for each technique and run. Populate the result in the following tabular formats:

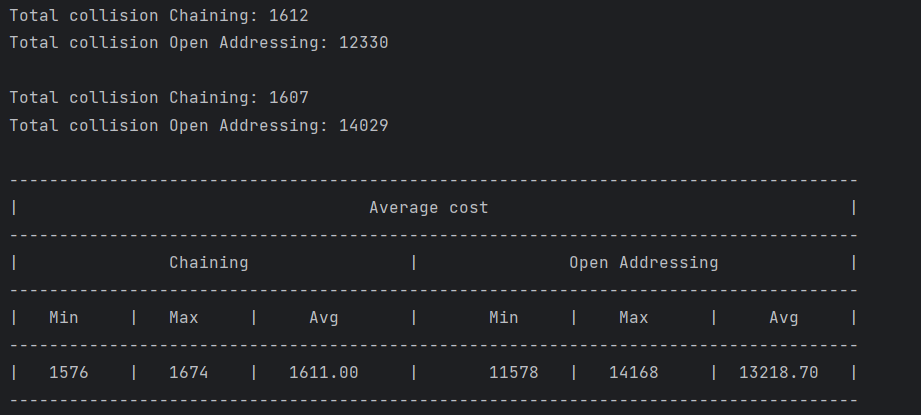
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Average Cost | | | | | |
| Chaining | | | Open Addressing | | |
| Minimum | Maximum | Average | Minimum | Maximum | Average |
|  |  |  |  |  |  |

Note: reminder to clean up the hash table before the next run.

Discuss your observation and finding.

### Output





### Discussion

Chaining's overall collision number ranges from 1576 to 1674, with an average of 1611. This number represents the number of collisions which took place when the chaining method was used to resolve collisions. In contrast, the total collision count for open addressing ranges from 11578 to 14168, with an average of 13218.7. The numbers signify the number of collisions that occurred when using the open addressing method. When comparing the chaining and open addressing, chaining generates less collisions. This is because of the reason that in chaining, each slot in the hash table may store numerous items in the format of linked lists, which allows effective collision handling.

However, open addressing has a higher number of collision because when a collision takes place in open addressing, the algorithm tries to discover the next available position by exploring the consecutive slots, potentially resulting in item clustering and more collisions. The amounts of collisions in open addressing can be influenced by aspects such as the load factor (the ratio of occupied spaces to total spaces) and the probing strategy employed (e.g., linear probing, quadratic probing, and so on). The arrangement and content of the input data could impact the amounts of collisions in open addressing. Particular trends in the input data can cause more collisions in open addressing than in chaining.

Chaining and open addressing both has its own strengths and weaknesses. Chaining may lead to increase in memory overhead because of the necessity for linked lists in each slot, which could contribute to higher memory utilisation. Furthermore, when long chains formed as resulting of high collision rates, performance degrades, impacting searching and inserting processes. Another disadvantage of chaining is the items in linked lists may not be stored consecutively, resulting in longer access times (Hashing - Open Addressing for Collision Handling, n.d.). Conversely, open addressing can result from increased clustering, which occurs when consecutive collisions lead items to be stored near together, which causes longer probe sequences and more collisions. High load factors can also make open addressing difficult since the amounts of collisions and probe sequences increases, affecting efficiency. Furthermore, open addressing necessitates a fixed-size database, making subsequent insertions difficult to manage without severe collisions or the requirement to enlarge the table (Hashing - Open Addressing for Collision Handling, n.d.). These problems show the importance of adopting a collision resolution approach for hash tables based on the individual requirements and characteristics of the data.

In summary, the outcomes imply that the chaining technique performs open addressing in the current scenario by having less collisions. However, considering the specific use case, the distribution of the input data, and the hash function selected, the efficiency of any collision resolution approach can change.

### Weakness of solution

The programme lacks suitable error handling techniques. It does not deal with exceptions or unanticipated circumstances that may occur during the execution, for example, memory failures, division by zero, or other potential problems. The stability as well as accuracy of the approach would increase with proper error handling.

The programme uses limited evaluation metrics which means that the output shows only the total number of collisions for each collision resolution technique. It lacks other evaluation metrics, like the load factor, average chain length (for chaining), and probing sequence analysis (for open addressing). These measurements can provide additional information into the effectiveness and efficacy of the collision resolution method.

Furthermore, the system lacks test coverage. The statistic simply indicates the average number of collisions for several executes of the programme. It fails to provide extensive testing or coverage of many sides of situations, such as entering duplicate keys, or evaluating the behaviour with varied load factors. Thorough testing would aid in the detection of any flaws and other case handling concerns.

Finally, there is also a lack of code optimisations in the code. There are no performance optimisations in the code, including employing data structures like linked lists for chaining or more effective probing sequences for open addressing. Optimisations may decrease the number of comparisons, increase memory utilisation, and improve the overall performance.

### Code References

(Simranjenny84, n.d.)

(EntilZha, 2015)

(Zaczyński, n.d.)

(stephengrice, 2018)

## Question 3: Graph

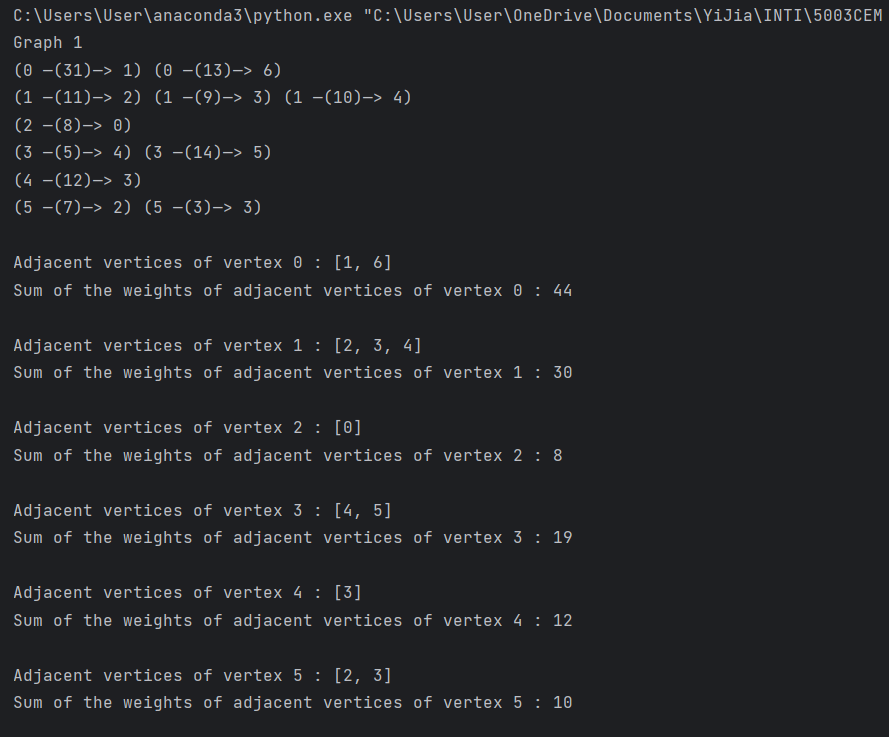
Write a program that uses graph concept. The program shall fulfil the following requirements:

* construct a weighted directed class graph that consists of the following methods:
* listAdjacentVertex: List all the adjacent vertex for a given vertex
* sumHighestAdjacentVertex: sum up all the weight for all the adjacent Vertices.
* Create two different graph objects (least 6 vertexes for each graph) and call the above two functions and display the results.

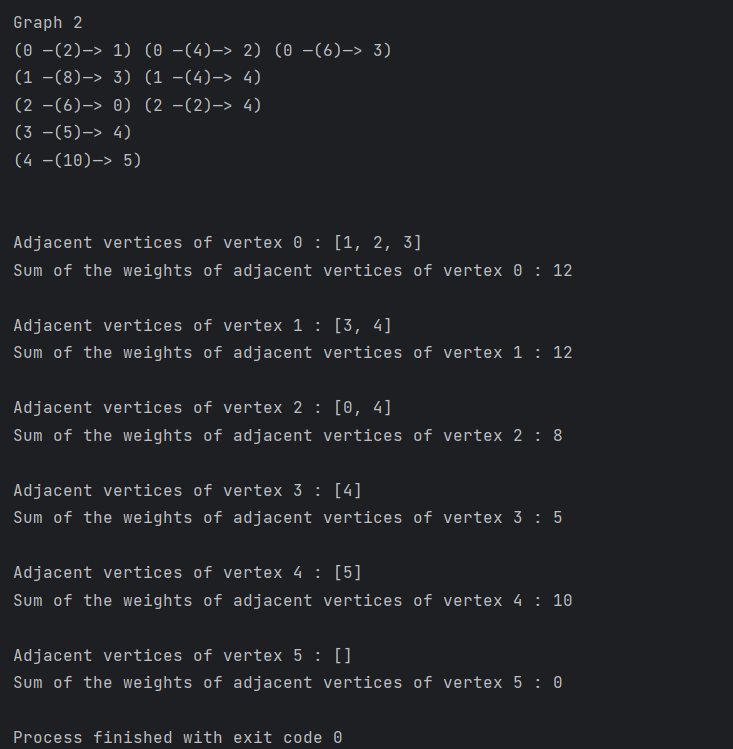
Explain the method/design you had used to present the graph structure in your program.

### Output

The printing adjacency list for graph 1 and its adjacent vertex with sum of the weight of the vertices.



The printing adjacency list for graph 2 and its adjacent vertex with sum of the weight of the vertices.



### Discussion

The code employs an adjacency list structure to implement a weighted directed graph. The WeightedGraph class contains methods for listing neighbouring vertices and calculating the total weights for adjacent vertices of a given vertex. An empty adjacency list of size n is initialised by allocating memory in the constructor method (\_\_init\_\_). It then loops through the list of edges given, adding each edge to the adjacency list. Every element in the adjacency list is a tuple (destination, weight), which represents the destination vertex and the edge weight.

The listAdjacentVertex function iterates over the adjacency list and extracts the adjacent vertices from the tuples to produce a list of adjacent vertices for a particular vertex.

The sumHighestAdjacentVertex function measures the sum of weights for every vertex's adjacent vertices. The function uses knowledge of lists to extract the weights from the tuples in the adjacency list after calling listAdjacentVertex to get the connected vertices. The function that returns the sum is then used to get the sum of all these weights.

The printWeightedGraph method is to print the graph's adjacency list representation. The function iterates over the adjacency list with nested loops, printing each vertex together with the neighbouring vertices and the weights.

Two graphs (graph1 and graph2) are created by applying the given edges in the \_\_main\_\_ section. The printWeightedGraph function is then used to print each graph's adjacency list representation. Finally, for every vertex within the graph, the neighbouring vertices are displayed by using the methods listAdjacentVertex and sumHighestAdjacentVertex for the sum of the weights of the adjacent vertices.

Overall, the code efficiently uses the adjacency list format and provides capabilities for connecting with the graph. It illustrates methods to build graph-related operations and delivers the basis for future graph-based algorithms and analysis.

### Weakness of solution

The code is presumptively written using a specified set of edges and weights in the form of lists. It lacks a way for dynamically inputting or modifying the graph structure during execution. Edges and weights are hard-coded within the programming. This constrains the programme's flexibility and reusability because it cannot accommodate alternative input graphs without altering the algorithm.

The printWeightedGraph function only displays the graph's adjacency list representation. Although this can help with visualisation, it might not be suitable for huge or complex graphs. Additional formatting or visualisation choices, such as displaying the graph with a graph visualisation library or providing more details, would improve the output's utility.

The code focuses on building the graph and giving methods for listing neighbouring vertices and calculating the sum of adjacent vertices' weights. Other commonly used graph operations, such as graph traversal techniques (BFS, DFS), shortest path methods (Dijkstra's, Bellman-Ford), and minimal spanning tree algorithms (Kruskal, Prim), are not included. As a result, its usefulness is limited, and it may not be suitable for more complicated graph-related tasks.

### Code References

(Graph Implementation in Python, n.d.)

(‘Graph Data Structure’, n.d.)

## Question 4: Concurrent process

(a) Write a function to calculate and display the car loan monthly repayment. Assume flat interest rate is used.

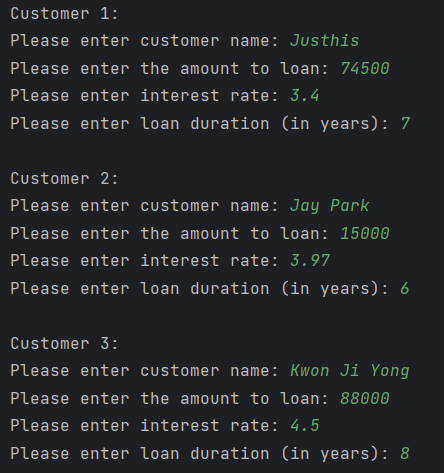
Refer to the link below to understand how to calculate the monthly repayment of a loan:

https://www.comparehero.my/personal-loan/articles/heres-how-car-loans-work-and-why-interest-charges-are-higher-than-you-think

(b) Write a program to allow users to calculate monthly repayment for 3 customers. The program shall call the function define in (a) concurrently.

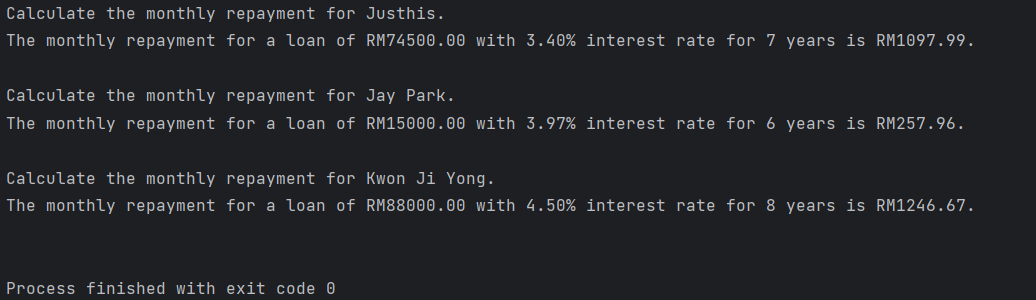
Observe the output of your solution. Discuss the theorical behind to support your observation in the documentation.

### Output



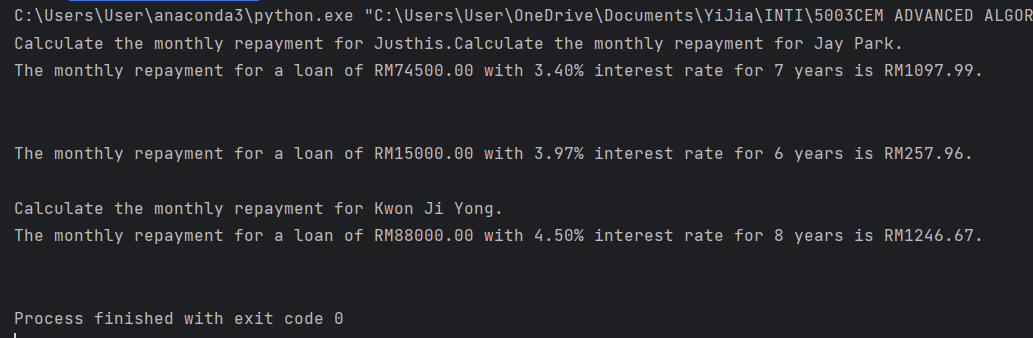
The system asked for user input to calculate the monthly repayment for 3 customers.

Output 1:



The output above shows that when the program is executed concurrently, it gives the expected output.

Output 2:



The output above illustrates that the monthly repayment calculations and displays for each customer are not in the expected order. This can be possible due to concurrent thread execution and the unpredictable nature of thread scheduling.

### Discussion

Based on the outputs above, the first result shows that the calculations and display are executed in the appropriate sequence, with each customer's repayment data being calculated and displayed separately before going on to the next customer. The sequential execution in the threading is most likely due to the structure of the operation, which performs every calculation and output action in a deterministic order, showing the expected behaviour when each of the threads is merged in the main programme once they have finished their calculations. This means concurrent execution does not take place in this case. Thus, race conditions or synchronization problems do not occur.

However, in the second output, the statement "Calculating monthly repayment" for Justhis appears first, merging with the statement for Jay Park. After that, the system only displays the statements of calculation for Justhis, followed by extra lines of spacing and the calculation statement for Jay Park. This shows that the calculations for Justhis and Jay Park are started at almost the same time, but the output for Justhis is displayed first and the output for Jay Park is delayed. Lastly, only the statement calculating the loan and the results of the calculation for Kwon Ji Yong are displayed. This output shows the non-deterministic behaviour of concurrent execution, where the sequence in which the threads finish their calculations and the display statements are synchronised can change. This is due to the nature of thread scheduling, in which separate threads can change at different rates depending on factors such as compute speed, system resource accessibility, and the thread scheduler's decisions. The order in which threads are processed is decided by the operating system's thread scheduler, and it can change within processes as well as during the same process. Therefore, the order in which outcomes are presented could differ because of the overlapping of thread execution. This is typical behaviour for concurrent execution and does not always point to a race condition or synchronisation problem.

In conclusion, the theoretical ideas of sequential execution and concurrent execution are apparent in the observable results. Output 1 shows sequential execution in threading, in which operations are finished in an ordered manner to ensure the intended outcome. Output 2 illustrates concurrent execution in threading, in which multiple processes are carried out concurrently, which might result in changes to the sequence in which they are completed and demonstrated.

### Weakness of solution

The code now determines monthly repayments for a group of customers of 3. It allows for user input to customise loan data but do not allows dynamically add new consumers. Adding user involvement via command-line inputs or a graphical user interface would improve the solution's usability and flexibility. Besides that, currently the code uses a set number of threads determined by the number of customers. If the number of customers substantially rises, this strategy might not scale properly. A thread pool or other thread management strategies might be used to efficiently handle a larger number of consumers.

Another weakness for the solution is insufficient calculation logic. The code just calculates the total repayment amount and does not take into account extra aspects like fees, insurance coverage, or other costs that may alter the overall loan repayment. Adding such components in the calculation logic would give an improved estimation of the monthly repayments.

Furthermore, threading might face the problem of synchronization or race condition when sharing the same resources. The code does not employ synchronisation methods such as mutex (lock) or semaphore to ensure thread safety while publishing results and progress. This implies that if many threads alter the shared resources such as customer list at the same time, it may result in unexpected performance or data corruption. To safeguard shared resources, it would be advantageous to use synchronisation techniques like locks or semaphores to ensure the accurate and consistent functioning of concurrent threads.

### Code References

(Brownlee, 2022)

(Multithreading in Python | Set 1, n.d.)

# Reflection

The main challenge that I faced in the coursework is when designing and implementing the data structure especially the binary search tree (BST) and hash table to fulfil the conditions is tough. It requires to have a deeper comprehension in data structure so that can achieve in implemented a better program to handle those situations. For example, the question 1 cake ordering system that implement using BST desires to spent a lot of time to implement the system because it has many values and data to save which requires many errors handling and testing to meet the requirements.

Besides that, the documentation after implemented all the program is also a challenge for me. This is because documentation involves a lot of explanation and elaborations on the theories, techniques used in the implementation. If does not have clear understanding on the theories or methods will not able to write the reports and discuss on the implementation for those topics especially the concurrent process.

On the other hand, this coursework provides many valuable experiences for me and have gained a lot of knowledge regarding the data structure and the algorithms that works in the computer. It allows me to practice on designing the program in data structure to solve the problem that may use in the future. I have understand more deeply in the programming, designing suitable algorithms or data structure on the situations, and problem-solving.

In conclude, although the coursework is quite challenging when solving the questions and needs to devote times in solving it. However, it was worthwhile because it helped me to improve my skills in many factors such as programming, data structures and algorithms, and also the thinking and problem-solving skills.

# References

Bhat, S. (n.d.) Deletion in Binary Search Tree [online] available from <https://www.geeksforgeeks.org/deletion-in-binary-search-tree/> [5 June 2023]

Brownlee, J. (2022) Python Threading: The Complete Guide [online] available from <https://superfastpython.com/threading-in-python/> [6 June 2023]

EntilZha (2015) Hash Table (Open Address) Implementation in Python Practicing for Interviews. available from <https://gist.github.com/EntilZha/5397c02dc6be389c85d8/revisions>

‘Graph Data Structure’ (n.d.) in BogoToBogo [online] available from <https://www.bogotobogo.com/python/python\_graph\_data\_structures.php> [5 June 2023]

Graph Implementation in Python (n.d.) available from <https://www.techiedelight.com/graph-implementation-python/> [5 June 2023]

Hashing - Open Addressing for Collision Handling (n.d.) available from <https://www.javatpoint.com/hashing-open-addressing-for-collision-handling> [5 June 2023]

Kumar, V. (2021) Delete a Node from a Binary Search Tree in Python [online] available from <https://www.codespeedy.com/delete-a-node-from-a-binary-search-tree-in-python/> [5 June 2023]

Lane, W. (2021) ‘Writing a Binary Search Tree in Python With Examples’. in Boot.Dev [online] available from <https://blog.boot.dev/computer-science/binary-search-tree-in-python/> [5 June 2023]

Multithreading in Python | Set 1 (n.d.) available from <https://www.geeksforgeeks.org/multithreading-python-set-1/> [6 June 2023]

PranchalK (n.d.) How to Implement Decrease Key or Change Key in Binary Search Tree? [online] available from <https://www.geeksforgeeks.org/how-to-implement-decrease-key-or-change-key-in-binary-search-tree/> [5 June 2023]

Rastogik346 (n.d.) Search and Insertion in Binary Search Tree [online] available from <https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/> [5 June 2023]

Simranjenny84 (n.d.) Implementation of Hashing with Chaining in Python [online] available from <https://www.geeksforgeeks.org/implementation-of-hashing-with-chaining-in-python/>

stephengrice (2018) Hashtable. available from <https://github.com/pagekeytech/education/tree/master/HashTable>

Zaczyński, B. (n.d.) Build a Hash Table in Python With TDD [online] available from <https://realpython.com/python-hash-table/>a

# Appendix

## Question 1

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

## Question 2

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

# Update minimum and maximum collisions for chaining  
# compares the current value of min or max with total\_collision\_chaining,  
# then selects the smaller of the two values and assigns it back

chaining\_min\_collisions = min(chaining\_min\_collisions, total\_collision\_chaining)  
chaining\_max\_collisions = max(chaining\_max\_collisions, total\_collision\_chaining)

# Update minimum and maximum collisions for open addressing  
 open\_addressing\_min\_collisions = min(open\_addressing\_min\_collisions, open\_addressing\_table.total\_collision\_open\_addressing)  
 open\_addressing\_max\_collisions = max(open\_addressing\_max\_collisions, open\_addressing\_table.total\_collision\_open\_addressing)  
  
 # Add the total collisions to calculate the average after every execution  
 chaining\_avg\_collisions += total\_collision\_chaining  
 open\_addressing\_avg\_collisions += open\_addressing\_table.total\_collision\_open\_addressing  
  
 # Clean up hash tables after every execution  
 chaining\_table.chaining\_table = [[] for \_ in range(table\_size)]  
 open\_addressing\_table.open\_addressing\_table = [None] \* table\_size  
  
 # Calculate average collisions  
 chaining\_avg\_collisions /= num\_execute # average = total average / 10  
 open\_addressing\_avg\_collisions /= num\_execute  
  
 # Print results in tabular format  
 print("-------------------------------------------------------------------------------------")  
 print("|\t\t\t\t\t\t\t\t\tAverage cost\t\t\t\t\t\t\t\t\t|")  
 print("-------------------------------------------------------------------------------------")  
 print("|\t\t\t\tChaining\t\t\t\t|\t\t\t\tOpen Addressing\t\t\t\t|")  
 print("-------------------------------------------------------------------------------------")  
 print("|\tMin\t\t|\tMax\t\t|\t Avg\t\t|\t\tMin\t\t|\t Max\t | \tAvg\t\t|")  
 print("-------------------------------------------------------------------------------------")  
 print(f"|\t{chaining\_min\_collisions}\t|\t{chaining\_max\_collisions}\t|\t{chaining\_avg\_collisions:.2f}\t\t|\t\t"  
 f"{open\_addressing\_min\_collisions}\t|\t{open\_addressing\_max\_collisions}\t | {open\_addressing\_avg\_collisions:.2f}\t|")  
 print("-------------------------------------------------------------------------------------")  
  
  
run\_program()

## Question 3

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

## Question 4

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