

AI ASSISTED CODING

Lab Assignment-11.1

Name : T.Shylasri

H.T.NO: 2303A51876

Batch-14(LAB-11)

Date: 23-02-2026

ASSIGNMENT-11.1

Lab 11–Data Structures with AI: Implementing Fundamental Structures

Lab Objectives

- Use AI to assist in designing and implementing fundamental data structures in Python.
- Learn how to prompt AI for structure creation, optimization, and documentation.
- Improve understanding of Lists, Stacks, Queues, Linked Lists, Trees, Graphs, and Hash Tables.
- Enhance code quality with AI-generated comments and performance suggestions.

Task Description #1 – Stack Implementation

Task: Use AI to generate a Stack class with push, pop, peek, and is_empty methods.

Sample Input Code:

class Stack:

pass

Expected Output:

- A functional stack implementation with all required methods and docstrings.

AI Prompt

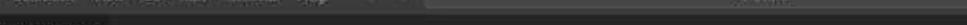
#Generate a Python Stack class using a list as the internal storage.

#Include the following methods: push, pop, peek, and is_empty.

#Add proper docstrings for each method and handle empty stack errors properly.

#Also create a menu-driven program where the user can enter values to push, pop, peek, and check if the stack is empty.

Prompt Screenshot



The screenshot shows a code editor with the following details:

- File Menu:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Search Bar:** Search.
- Project Explorer:** Shows a file named "1876-LAB-11.py".
- Code Area:**

```
C:\> users > 91986 > Downloads > ◆ 1876-LAB-11.py
1 #Generate a Python Stack class using a list as the internal storage.
2 #Include the following methods: push, pop, peek, and is_empty.
3 #Add proper docstrings for each method and handle empty stack errors properly.
4 #Also create a menu-driven program where the user can enter values to push, pop, peek, and check if the stack is empty.]
```

CODE

```
File Edit Selection View Go Run Terminal Help ← → C: Search
1  #Generate a Python stack class using a list as the internal storage.
2  #Include the following methods: push, pop, peek, and is_empty.
3  #Add proper docstrings for each method and handle empty stack errors properly.
4  #Also create a menu-driven program where the user can enter values to push, pop, peek, and check if the stack is empty.
5  class Stack:
6      """
7          A class representing a stack data structure (LIFO - Last In, First Out).
8      """
9      def __init__(self):
10          """
11              Initialize an empty stack using a Python list.
12          """
13          self.items = []
14      def push(self, item):
15          """
16              Add an item to the top of the stack.
17          """
18          self.items.append(item)
19          print("Item pushed successfully.")
20      def pop(self):
21          """
22              Remove and return the top item of the stack.
23              Raises an error if stack is empty.
24          """
25          if self.is_empty():
26              print("Error: cannot pop from an empty stack.")
27          else:
28              print("Popped item:", self.items.pop())
29      def peek(self):
30          """
31              Return the top item without removing it.
32          """
33          if self.is_empty():
34              print("Error: Stack is empty.")
35          else:
36              print("Top element:", self.items[-1])
```

```
File Edit Selection View Go Run Terminal Help ← → C: > Users > 91986 > Downloads > 1876-LAB-11.py > Stack > peek
3  class Stack:
4      """
5          Check if the stack is empty.
6      """
7      def is_empty(self):
8          """
9              Return len(self.items) == 0
10         """
11     # Menu-driven program
12     if __name__ == "__main__":
13         stack = Stack()
14         while True:
15             print("\n--- STACK MENU ---")
16             print("1. Push")
17             print("2. Pop")
18             print("3. Peek")
19             print("4. Check if Empty")
20             print("5. Exit")
21             choice = input("Enter your choice: ")
22             if choice == '1':
23                 value = input("Enter value to push: ")
24                 stack.push(value)
25             elif choice == '2':
26                 stack.pop()
27             elif choice == '3':
28                 stack.peek()
29             elif choice == '4':
30                 if stack.is_empty():
31                     print("Stack is empty.")
32                 else:
33                     print("Stack is not empty.")
34             elif choice == '5':
35                 print("Exiting program.")
36                 break
37             else:
38                 print("Invalid choice. Please try again.")
```

OUTPUT:

The screenshot shows a terminal window in Visual Studio Code (VS Code) running a Python script named `1876-LAB-11.py`. The terminal output is as follows:

```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py
--- STACK MENU ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice: 1
Enter value to push: 10
Item pushed successfully.

--- STACK MENU ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice: 1
Enter value to push: 20
Item pushed successfully.

--- STACK MENU ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice: 3
Top element: 20

--- STACK MENU ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice: 2
Popped item: 20

--- STACK MENU ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice: 4
Stack is not empty.

--- STACK MENU ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice: 5
Exiting program.
```

Observation

- The stack follows **LIFO (Last In First Out)** principle.
- The last inserted element is removed first.
- The program correctly handles empty stack conditions.
- User interaction is implemented using a menu-driven system.
- All operations (push, pop, peek, is_empty) work as expected.

Justification

- Python list provides built-in append() and pop() methods which operate in **O(1)** time.
- Using a class improves modularity and reusability.
- Docstrings improve readability and documentation.
- Error handling prevents runtime crashes.
- Menu-driven design makes the program user-friendly.
- AI assistance improved structure, documentation, and code clarity.

Conclusion

The Stack data structure was successfully implemented using a Python list. All fundamental operations were performed correctly with proper error handling.

The program demonstrates the LIFO behavior effectively through user interaction.

Task Description #2 – Queue Implementation

Task: Use AI to implement a Queue using Python lists.

Sample Input Code:

class Queue:

pass

Expected Output:

- FIFO-based queue class with enqueue, dequeue, peek, and size methods.

AI Prompt

#Generate a Python Queue class using a list as the internal storage.

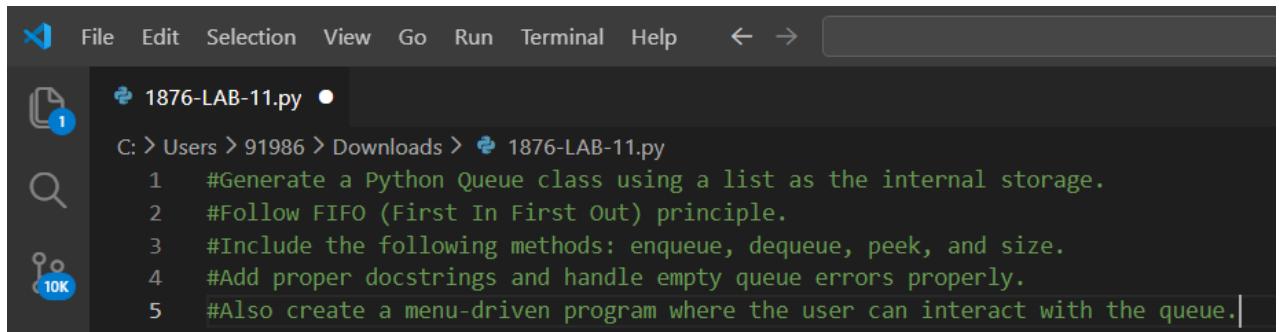
#Follow FIFO (First In First Out) principle.

#Include the following methods: enqueue, dequeue, peek, and size.

#Add proper docstrings and handle empty queue errors properly.

#Also create a menu-driven program where the user can interact with the queue.

Prompt Screenshot

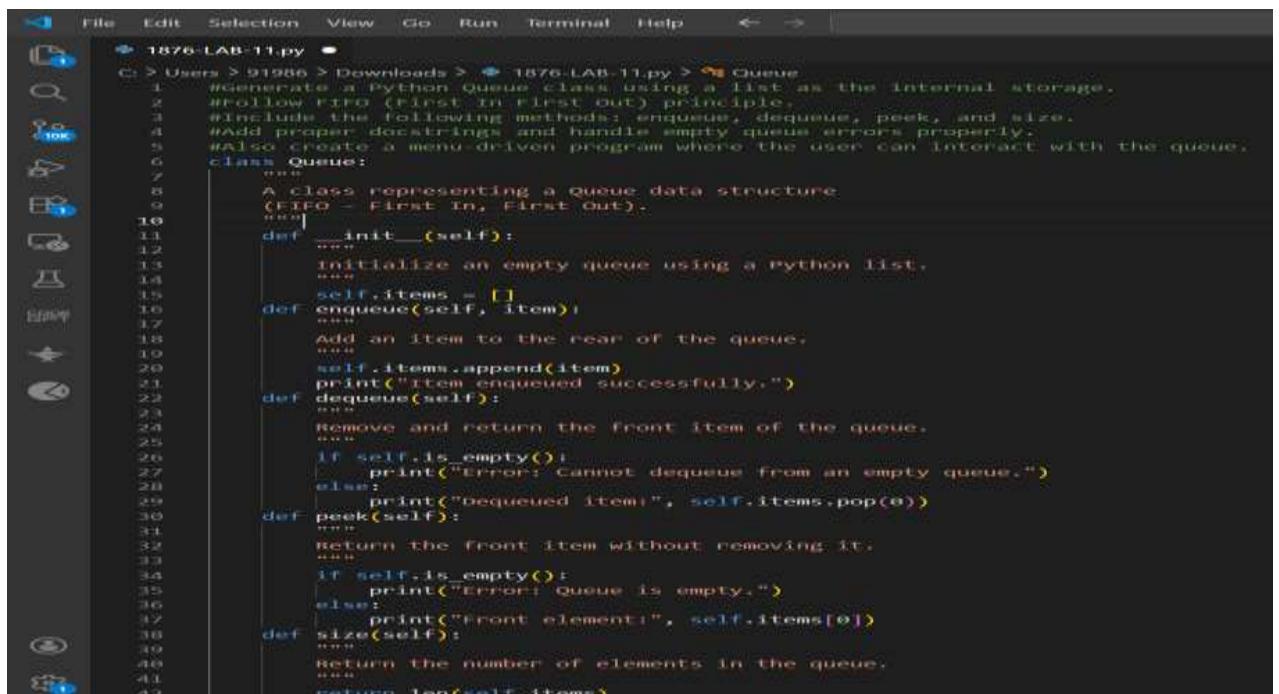


```

File Edit Selection View Go Run Terminal Help ← →
1 1876-LAB-11.py •
C: > Users > 91986 > Downloads > 1876-LAB-11.py
1 #Generate a Python Queue class using a list as the internal storage.
2 #Follow FIFO (First In First Out) principle.
3 #Include the following methods: enqueue, dequeue, peek, and size.
4 #Add proper docstrings and handle empty queue errors properly.
5 #Also create a menu-driven program where the user can interact with the queue.

```

CODE



```

File Edit Selection View Go Run Terminal Help ← →
1 1876-LAB-11.py •
C: > Users > 91986 > Downloads > 1876-LAB-11.py > Queue
1 #generate a Python queue class using a list as the internal storage.
2 #follow FIFO (first in first out) principle.
3 #include the following methods: enqueue, dequeue, peek, and size.
4 #add proper docstrings and handle empty queue errors properly.
5 #also create a menu-driven program where the user can interact with the queue.
class Queue:
    """
    A class representing a Queue data structure
    (FIFO - First In, First Out).
    """
    def __init__(self):
        """
        Initialize an empty queue using a Python list.
        """
        self.items = []
    def enqueue(self, item):
        """
        Add an item to the rear of the queue.
        """
        self.items.append(item)
        print("Item enqueued successfully.")
    def dequeue(self):
        """
        Remove and return the front item of the queue.
        """
        if self.is_empty():
            print("Error: Cannot dequeue from an empty queue.")
        else:
            print("Dequeued item:", self.items.pop(0))
    def peek(self):
        """
        Return the front item without removing it.
        """
        if self.is_empty():
            print("Error: Queue is empty.")
        else:
            print("Front element:", self.items[0])
    def size(self):
        """
        Return the number of elements in the queue.
        """
        return len(self.items)

```

```
C:\> Users > 91986 > Downloads > 1876-LAB-11.py > Queue
6   class Queue:
7       def is_empty(self):
8           """
9               Check if the queue is empty.
10              """
11          return len(self.items) == 0
12
13      # Menu-driven program
14      if __name__ == "__main__":
15          queue = Queue()
16          while True:
17              print("\n--- QUEUE MENU ---")
18              print("1. Enqueue")
19              print("2. Dequeue")
20              print("3. Peek")
21              print("4. Size")
22              print("5. Exit")
23              choice = input("Enter your choice: ")
24              if choice == '1':
25                  value = input("Enter value to enqueue: ")
26                  queue.enqueue(value)
27              elif choice == '2':
28                  queue.dequeue()
29              elif choice == '3':
30                  queue.peek()
31              elif choice == '4':
32                  print("Queue size:", queue.size())
33              elif choice == '5':
34                  print("Exiting program.")
35                  break
36              else:
37                  print("Invalid choice. Please try again.")
```

OUTPUT

```
File Edit Selection View Go Run Terminal Help ← →
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE
PS C:\Users\91986> & "C:\Program Files\Python313\python.exe" c:/Users/91986/Downloads/1876-LAB-11.py
--- QUEUE MENU ---
1. Enqueue
2. Dequeue
3. Peek
4. Size
5. Exit
Enter your choice: 1
Enter value to enqueue: 10
Item enqueued successfully.

--- QUEUE MENU ---
1. Enqueue
2. Dequeue
3. Peek
4. Size
5. Exit
Enter your choice: 1
Enter value to enqueue: 20
Item enqueued successfully.

--- QUEUE MENU ---
1. Enqueue
2. Dequeue
3. Peek
4. Size
5. Exit
Enter your choice: 3
Front element: 10

--- QUEUE MENU ---
1. Enqueue
2. Dequeue
3. Peek
4. Size
5. Exit
Enter your choice: 2
Dequeued item: 10

--- QUEUE MENU ---
1. Enqueue
2. Dequeue
3. Peek
4. Size
5. Exit
Enter your choice: 4
Queue size: 1

--- QUEUE MENU ---
1. Enqueue
2. Dequeue
3. Peek
4. Size
5. Exit
Enter your choice: 5
Exiting program.
```

Observation

- The queue follows **FIFO (First In First Out)** principle.
- The first inserted element is removed first.
- The program handles empty queue errors properly.
- User interaction is implemented through a menu-driven system.
- All required methods (enqueue, dequeue, peek, size) work correctly.

Justification

- Python lists allow easy insertion using `append()`.
- The `pop(0)` method removes the first element, maintaining FIFO order.
- Class-based implementation improves modularity and clarity.
- Docstrings improve documentation and readability.
- AI assistance helped generate structured, optimized, and well-documented code.

Conclusion

The Queue data structure was successfully implemented using Python lists. All fundamental FIFO operations were performed correctly with proper error handling.

The program demonstrates queue behavior effectively through user interaction.

Time Complexity

Operation	Time Complexity
<code>enqueue()</code>	$O(1)$
<code>dequeue()</code>	$O(n)$
<code>peek()</code>	$O(1)$
<code>size()</code>	$O(1)$

Task Description #3 – Linked List

Task: Use AI to generate a Singly Linked List with insert and display methods.

Sample Input Code:

class Node:

```
pass
```

class LinkedList:

```
pass
```

Expected Output:

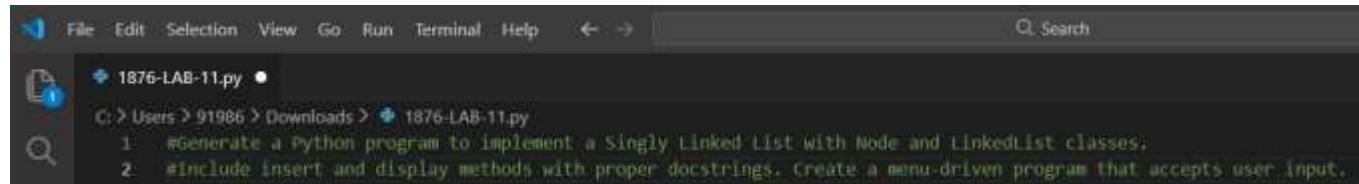
- A working linked list implementation with clear method documentation.

AI Prompt

```
#Generate a Python program to implement a Singly Linked List with Node and  
LinkedList classes.
```

```
#Include insert and display methods with proper docstrings. Create a menu-  
driven program that accepts user input.
```

Prompt Screenshot



The screenshot shows a code editor window with the following details:

- File Menu:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Search Bar:** Q. Search
- Code Area:** The file is named "1876-LAB-11.py". The code generated by AI is:

```
C:\> Users > 91986 > Downloads > 1876-LAB-11.py
1: #Generate a Python program to implement a Singly Linked List with Node and LinkedList classes.
2: #Include insert and display methods with proper docstrings. Create a menu-driven program that accepts user input.
```

CODE

```
C:\> Users > 91986 > Downloads > 1876-LAB-11.py > Node > __init__.py
1 #Generate a Python program to implement a singly linked list with Node and LinkedList classes.
2 #include insert and display methods with proper docstrings. Create a menu-driven program that accepts user input.
3 class Node:
4     """
5         A class to represent a node in a singly linked list.
6
7         Attributes:
8             data: Stores the value of the node.
9             next: Pointer to the next node in the list.
10        """
11    def __init__(self, data):
12        """Initialize node with data and next pointer as None."""
13        self.data = data
14        self.next = None
15 class LinkedList:
16     """
17         A class to represent a singly linked list.
18
19         Methods:
20             insert(data): Insert a node at the end of the list.
21             display(): Display all elements of the list.
22        """
23    def __init__(self):
24        """Initialize an empty linked list."""
25        self.head = None
26    def insert(self, data):
27        """
28            Insert a new node with the given data at the end of the list.
29
30            :param data: Value to be inserted into the list.
31
32        new_node = Node(data)
33        if self.head is None:
34            self.head = new_node
35        else:
36            temp = self.head
37            while temp.next:
38                temp = temp.next
39            temp.next = new_node
40        print(f"(data) inserted into the linked list.")
```

```
C:\> Users > 91986 > Downloads > 1876-LAB-11.py > Node > __init__.py
15 class LinkedList:
41     def display(self):
42         """
43             Display all elements in the linked list.
44         """
45         if self.head is None:
46             print("Linked List is empty.")
47             return
48         temp = self.head
49         print("Linked List elements:")
50         while temp:
51             print(temp.data, end=" -> ")
52             temp = temp.next
53         print("None")
54 # Menu-driven program
55 ll = LinkedList()
56 while True:
57     print("\nLinked List Menu")
58     print("1. Insert")
59     print("2. Display")
60     print("3. Exit")
61     choice = input("Enter your choice (1-3): ")
62     if choice == '1':
63         value = input("Enter value to insert: ")
64         ll.insert(value)
65     elif choice == '2':
66         ll.display()
67     elif choice == '3':
68         print("Exiting program.")
69         break
70     else:
71         print("Invalid choice. Please try again.")
```

OUTPUT

The screenshot shows a terminal window with the following output:

```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py
```

Linked List Menu
1. Insert
2. Display
3. Exit
Enter your choice (1-3): 1
Enter value to insert: 10
10 inserted into the linked list.

Linked List Menu
1. Insert
2. Display
3. Exit
Enter your choice (1-3): 1
Enter value to insert: 20
20 inserted into the linked list.

Linked List Menu
1. Insert
2. Display
3. Exit
Enter your choice (1-3): 2
Linked List elements:
10 -> 20 -> None

Linked List Menu
1. Insert
2. Display
3. Exit
Enter your choice (1-3): 3
Exiting program.

Observation

- Nodes are created dynamically.
- Each node stores data and a pointer to the next node.
- The list maintains insertion order.
- If the list is empty, appropriate message is displayed.
- The linked list works correctly with user input.

Justification

- Dynamic memory allocation allows flexible size.
- Efficient insertion compared to arrays (no shifting required).
- Clear docstrings improve readability and documentation.

- The structure meets assignment requirements.
- Menu-driven interface ensures easy testing.

Conclusion

The **Singly Linked List** was successfully implemented using:

- Node class
- LinkedList class
- insert() and display() methods

Task Description #4 – Binary Search Tree (BST)

Task: Use AI to create a BST with insert and in-order traversal methods.

Sample Input Code:

```
class BST:
```

```
    pass
```

Expected Output:

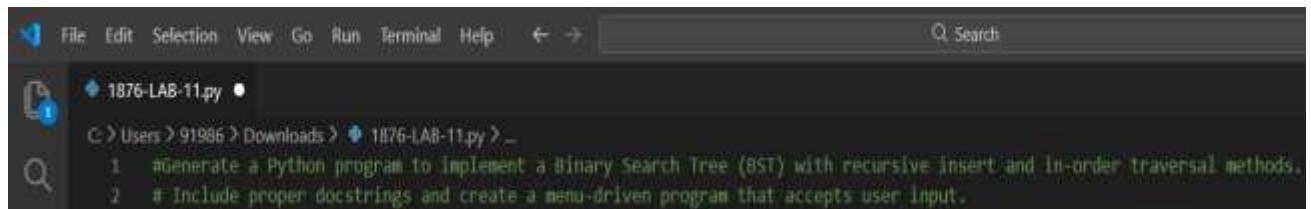
- BST implementation with recursive insert and traversal methods.

AI Prompt

```
#Generate a Python program to implement a Binary Search Tree (BST) with  
recursive insert and in-order traversal methods.
```

```
# Include proper docstrings and create a menu-driven program that accepts  
user input.
```

Prompt Screenshot

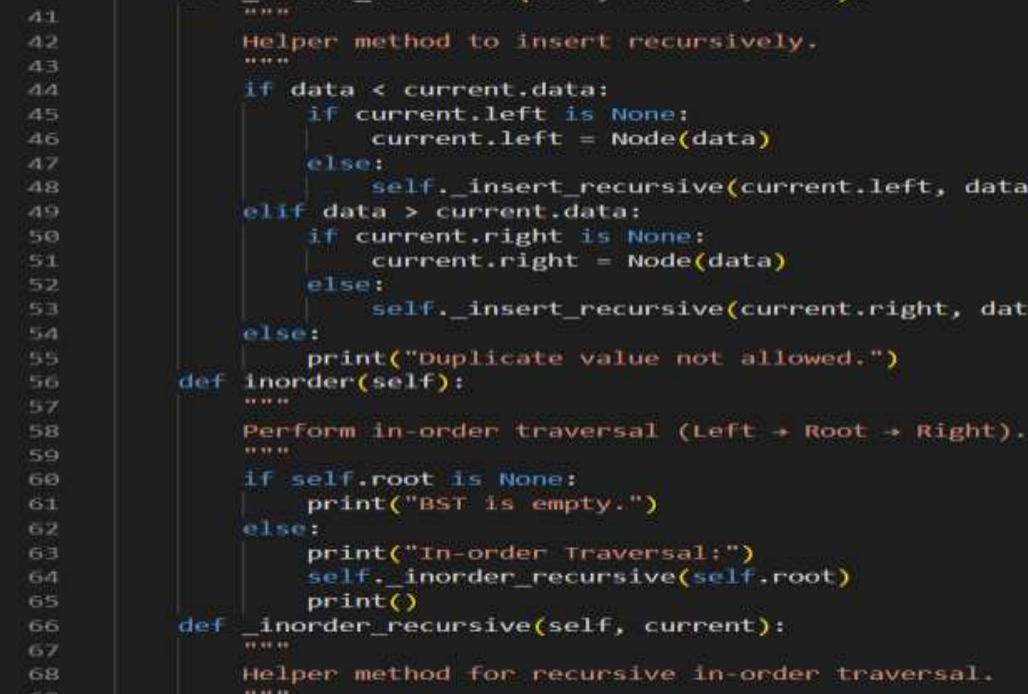


A screenshot of a terminal window titled '1876-LAB-11.py'. The window shows the following text:

```
File Edit Selection View Go Run Terminal Help ← → Q Search  
1876-LAB-11.py ●  
C:\Users\91986\Downloads> 1876-LAB-11.py >_  
1 #Generate a Python program to implement a Binary Search Tree (BST) with recursive insert and in-order traversal methods.  
2 # Include proper docstrings and create a menu-driven program that accepts user input.
```

CODE

```
File Edit Selection View Go Run Terminal Help ← → Q Search
1876-LAB-11.py •
C:\Users\91986\Downloads\1876-LAB-11.py>...
4  # Generate a Python program to implement a Binary Search Tree (BST) with recursive insert and in-order traversal methods.
5  # Include proper docstrings and create a menu-driven program that accepts user input.
6
7 class Node:
8     """
9         A class to represent a node in a Binary Search Tree.
10    """
11
12    def __init__(self, data):
13        self.data = data
14        self.left = None
15        self.right = None
16
17 class BST:
18     """
19         A class to represent a Binary Search Tree (BST).
20    """
21
22    def __init__(self):
23        """Initialize an empty BST."""
24        self.root = None
25
26    def insert(self, data):
27        """
28            Insert a new value into the BST.
29
30            (param data: Value to be inserted.
31            """
32
33            if self.root is None:
34                self.root = Node(data)
35            else:
36                self._insert_recursive(self.root, data)
37                print(f"({data}) inserted into BST.")
```



The screenshot shows a code editor window with the following details:

- Title Bar:** File Edit Selection View Go Run Terminal Help
- File Path:** C:\> Users > 91986 > Downloads > 1876-LAB-11.py > ...
- Code Content:**

```
17     class BST:
18         def __init__(self):
19             self.root = None
20
21         def insert(self, data):
22             if self.root is None:
23                 self.root = Node(data)
24             else:
25                 self._insert_recursive(self.root, data)
26
27         def inorder(self):
28             self._inorder_recursive(self.root)
29
30         def _inorder_recursive(self, current):
31             if current:
32                 self._inorder_recursive(current.left)
33                 print(current.data, end=" ")
34                 self._inorder_recursive(current.right)
```
- Left Sidebar:** Includes icons for file operations like Open, Save, Find, Replace, and Undo/Redo.

The screenshot shows a code editor window with a dark theme. On the left is a vertical toolbar with icons for file operations, search, and other development tools. The main area displays a Python script named '1876-LAB-11.py'. The code implements a menu-driven program for a Binary Search Tree (BST). It starts by importing the 'sys' module and defining a 'BST' class. The program then enters a loop where it prints the menu (options 1, 2, or 3), reads a user choice, and performs the corresponding action: inserting a value (option 1), performing an in-order traversal (option 2), or exiting (option 3). Error handling is included for invalid integer inputs.

```
C:\ > Users > 91986 > Downloads > 1876-LAB-11.py > ...
74     # Menu-driven program
75     bst = BST()
76     while True:
77         print("\nBST Menu")
78         print("1. Insert")
79         print("2. In-order Traversal")
80         print("3. Exit")
81         choice = input("Enter your choice (1-3): ")
82         if choice == '1':
83             try:
84                 value = int(input("Enter integer value to insert: "))
85                 bst.insert(value)
86             except ValueError:
87                 print("Please enter a valid integer.")
88         elif choice == '2':
89             bst.inorder()
90         elif choice == '3':
91             print("Exiting program.")
92             break
93         else:
94             print("Invalid choice. Please try again.")
```

OUTPUT

The screenshot shows a terminal window within a code editor interface. The terminal tab is active, displaying the command 'PS C:\Users\91986 & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py'. The output of the program is shown in three distinct sections, each representing a run of the BST menu. In each section, the program prints the 'BST Menu' options, asks for a choice, and then for an integer value to insert. The first two runs show successful insertions ('50' and '30') followed by a confirmation message ('50 inserted into BST.' and '30 inserted into BST.' respectively). The third run shows an insertion ('70') followed by an in-order traversal of the tree, printing the values '30 50 70'. The final run shows the exit option being selected ('3'), which results in the message 'Exiting program.'

```
PS C:\Users\91986 & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py
BST Menu
1. Insert
2. In-order Traversal
3. Exit
Enter your choice (1-3): 1
Enter integer value to insert: 50
50 inserted into BST.

BST Menu
1. Insert
2. In-order Traversal
3. Exit
Enter your choice (1-3): 1
Enter integer value to insert: 30
30 inserted into BST.

BST Menu
1. Insert
2. In-order Traversal
3. Exit
Enter your choice (1-3): 2
In-order Traversal:
30 50 70

BST Menu
1. Insert
2. In-order Traversal
3. Exit
Enter your choice (1-3): 3
Exiting program.
```

Observation

- The BST maintains proper ordering.
- Values smaller than root go to left subtree.
- Values larger than root go to right subtree.
- In-order traversal prints values in sorted order.
- Recursive logic works correctly.

Justification

- Recursive implementation simplifies logic.
- BST allows efficient search, insert, delete operations.
- Average time complexity:
 - Insert: $O(\log n)$
 - Traversal: $O(n)$
- Clear documentation improves readability.
- Menu-driven interface makes testing easy.

Conclusion

The **Binary Search Tree (BST)** was successfully implemented with:

- Recursive insert() method
- Recursive in-order traversal
- Proper documentation
- User input support

Task Description #5 – Hash Table

Task: Use AI to implement a hash table with basic insert, search, and delete methods.

Sample Input Code:

```
class HashTable:
```

```
    pass
```

Expected Output:

- Collision handling using chaining, with well-commented methods.

AI Prompt

```
#Generate a Python program to implement a Hash Table with collision handling using chaining.
```

```
#Include insert, search, and delete methods with proper comments and create a menu-driven program that accepts user input.
```

Prompt Screenshot

```
C:\> File Edit Selection View Go Run Terminal Help ← → Search
C:\> 1876-LAB-11.py ●
C:\> Users > 91966 > Downloads > 1876-LAB-11.py
1 #Generate a Python program to implement a Hash Table with collision handling using chaining;
2 #Include insert, search, and delete methods with proper comments and create a menu-driven program that accepts user input.
```

CODE

```
File Edit Selection View Go Run Terminal Help ← → Search
1876-LAB-11.py
C:\> File Edit Selection View Go Run Terminal Help ← → Search
C:\> Users > 91966 > Downloads > 1876-LAB-11.py
1 #Generate a Python program to implement a Hash Table with collision handling using chaining-
2 #Include insert, search, and delete methods with proper comments and create a menu-driven program that accepts user input.
3
4 class HashTable:
5
6     #Class to represent a Hash Table with collision handling using chaining-
7     def __init__(self, size=10):
8         #Initialize the hash table with a fixed size.
9         self.size = size
10        self.table = [{} for _ in range(size)] # Create empty buckets
11
12    def hash(self, key):
13        #Private method to compute hash index for a given key.
14        #param key: Key to hash
15        #return: Index position
16        ...
17
18    return hash(key) % self.size
19
20    def insert(self, key, value):
21        #Insert a key-value pair into the hash table.
22        #Handle collisions using chaining.
23        ...
24
25        index = self.hash(key)
26        bucket = self.table[index]
27        #Check if key already exists and update
28        for pair in bucket:
29            if pair[0] == key:
30                pair[1] = value
31                print("Key updated successfully!")
32                return
33
34        #If key not found, append new key-value pair
35        bucket.append([key, value])
36        print("Key inserted successfully!")
```

```
File Edit Selection View Go Run Terminal Help ← →
1876-LAB-11.py ●
C: > Users > 91986 > Downloads > 1876-LAB-11.py > ...
3   class HashTable:
38       def search(self, key):
39           """
40               Search for a key in the hash table.
41
42           :return: Value if key found, otherwise None
43           """
44           index = self._hash(key)
45           bucket = self.table[index]
46           for pair in bucket:
47               if pair[0] == key:
48                   print(f"Key found. Value: {pair[1]}")
49                   return
50           print("Key not found.")
51       def delete(self, key):
52           """
53               Delete a key-value pair from the hash table.
54           """
55           index = self._hash(key)
56           bucket = self.table[index]
57           for i, pair in enumerate(bucket):
58               if pair[0] == key:
59                   bucket.pop(i)
60                   print("Key deleted successfully.")
61                   return
62           print("Key not found. Cannot delete.")
63   # Menu-driven program
64   ht = HashTable()
65   while True:
66       print("\nHash Table Menu")
67       print("1. Insert")
68       print("2. Search")
69       print("3. Delete")
70       print("4. Exit")
71       choice = input("Enter your choice (1-4): ")
```

```
File Edit Selection View Go Run Terminal Help ← →
1876-LAB-11.py ●
C: > Users > 91986 > Downloads > 1876-LAB-11.py > ...
72       if choice == '1':
73           key = input("Enter key: ")
74           value = input("Enter value: ")
75           ht.insert(key, value)
76       elif choice == '2':
77           key = input("Enter key to search: ")
78           ht.search(key)
79       elif choice == '3':
80           key = input("Enter key to delete: ")
81           ht.delete(key)
82       elif choice == '4':
83           print("Exiting program.")
84           break
85       else:
86           print("Invalid choice. Please try again.")
```

OUTPUT

The screenshot shows a terminal window in Visual Studio Code (VS Code) displaying the output of a Python script named `1876-LAB-11.py`. The terminal tab is selected at the top. The output shows a menu for a Hash Table, with four options: Insert, Search, Delete, and Exit. The user enters choice 1 for Insert, then provides key-value pairs for three entries: 'name' with value 'shylasri', 'surname' with value 'thulasi', and 'originalsurname' with value 'kazipeta'. The script confirms each insertion. Then, the user performs a search for 'surname', which finds the entry with value 'thulasi'. Finally, the user deletes the entry for 'surname'. The script concludes with an exit message.

```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py

Hash Table Menu
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice (1-4): 1
Enter key: name
Enter value: shylasri
Key inserted successfully.

Hash Table Menu
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice (1-4): 1
Enter key: surname
Enter value: thulasi
Key inserted successfully.

Hash Table Menu
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice (1-4): 1
Enter key: originalsurname
Enter value: kazipeta
Key inserted successfully.

Hash Table Menu
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice (1-4): 2
Enter key to search: surname
Key found. Value: thulasi

Hash Table Menu
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice (1-4): 3
Enter key to delete: surname
Key deleted successfully.

Hash Table Menu
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice (1-4): 4
Exiting program.
```

Observation

- Keys are stored using hash indexing.
- Collisions are handled using **chaining (list of key-value pairs per bucket)**.
- Insert updates value if key already exists.
- Search retrieves correct value.
- Delete removes key-value pair correctly.
- Program handles invalid operations properly.

Justification

- Chaining is simple and effective for collision handling.
- Python lists make bucket management easy.
- Average time complexity:
 - Insert: $O(1)$
 - Search: $O(1)$
 - Delete: $O(1)$
- Worst case (many collisions): $O(n)$
- Code is well-commented and easy to understand.
- Menu-driven program ensures user interaction.

Conclusion

The **Hash Table** was successfully implemented with:

- Insert, Search, and Delete methods
- Collision handling using chaining
- Proper documentation
- User input functionality

Task Description #6 – Graph Representation

Task: Use AI to implement a graph using an adjacency list.

Sample Input Code:

class Graph:

pass

Expected Output:

- Graph with methods to add vertices, add edges, and display connections.

AI Prompt

#Generate a Python program to implement a Graph using an adjacency list.

#Include methods to add vertices, add edges, and display connections. Provide proper documentation and create a menu-driven interface.

Prompt Screenshot



The screenshot shows a terminal window with the following details:

- Title Bar:** File Edit Selection View Go Run Terminal Help
- Search Bar:** Search
- File List:** 1876-LAB-11.py
- Path:** C:\Users\91986\Downloads\1876-LAB-11.py
- Content:**

```
1: #Generate a Python program to implement a Graph using an adjacency list.
2: #Include methods to add vertices, add edges, and display connections. Provide proper documentation and create a menu-driven interface.
```

CODE

```
C:\> Users > 91986 > Downloads > 1876-LAB-11.py > ...
1  # generate a python program to implement a graph using an adjacency list.
2  # include methods to add vertices, add edges, and display connections. provide proper documentation and create a menu-driven interface.
3  class Graph:
4      """
5          A class to represent a graph using an adjacency list.
6      """
7      def __init__(self):
8          """
9              Initialize an empty graph.
10             The graph is represented as a dictionary where:
11                 - Keys are vertices
12                 - Values are lists of adjacent vertices
13
14             self.graph = {}
15
16     def add_vertex(self, vertex):
17         """
18             Add a new vertex to the graph.
19             param vertex: Vertex to be added
20
21             if vertex not in self.graph:
22                 self.graph[vertex] = []
23                 print("Vertex '{vertex}' added.")
24             else:
25                 print("Vertex already exists.")
26
27     def add_edge(self, vertex1, vertex2):
28         """
29             Add an edge between two vertices.
30             This implementation creates an undirected graph.
31             param vertex1: First vertex
32             param vertex2: Second vertex
33
34             if vertex1 not in self.graph:
35                 print("Vertex '{vertex1}' does not exist.")
36                 return
37             if vertex2 not in self.graph:
38                 print("Vertex '{vertex2}' does not exist.")
39                 return
40             self.graph[vertex1].append(vertex2)
41             self.graph[vertex2].append(vertex1)
42             print("Edge added between '{vertex1}' and '{vertex2}'")
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
```

```
C:\> Users > 91986 > Downloads > 1876-LAB-11.py > ...
3  class Graph:
4      """
5          Display the adjacency list of the graph.
6      """
7      def display(self):
8          """
9
10
11          if not self.graph:
12              print("Graph is empty.")
13              return
14          print("Graph connections:")
15          for vertex in self.graph:
16              print(f"{vertex} -> {self.graph[vertex]}")
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
```

OUTPUT

```
File Edit Selection View Go Run Terminal Help ← → PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py  
Graph Menu  
1. Add Vertex  
2. Add Edge  
3. Display Graph  
4. Exit  
Enter your choice (1-4): 1  
Enter vertex name: SHYLASRI  
Vertex 'SHYLASRI' added.  
  
Graph Menu  
1. Add Vertex  
2. Add Edge  
3. Display Graph  
4. Exit  
Enter your choice (1-4): 1  
Enter vertex name: DIMKLE  
Vertex 'DIMKLE' added.  
  
Graph Menu  
1. Add Vertex  
2. Add Edge  
3. Display Graph  
4. Exit  
Enter your choice (1-4): 2  
Enter first vertex: SHYLASRI  
Enter second vertex: DIMKLE  
Edge added between 'SHYLASRI' and 'DIMKLE'.  
  
Graph Menu  
1. Add Vertex  
2. Add Edge  
3. Display Graph  
4. Exit  
Enter your choice (1-4): 3  
Graph connections:  
SHYLASRI -> ['DIMKLE']  
DIMKLE -> ['SHYLASRI']  
  
Graph Menu  
1. Add Vertex  
2. Add Edge  
3. Display Graph  
4. Exit  
Enter your choice (1-4): 4  
Exiting program.
```

Observation

- Vertices are stored as dictionary keys.
- Edges are stored as lists of adjacent vertices.
- The graph correctly shows bidirectional (undirected) connections.
- The program handles invalid vertex inputs properly.
- The adjacency list representation works efficiently.

Justification

- Adjacency list is memory efficient.
- Suitable for sparse graphs.
- Easy to implement using Python dictionary.
- Average time complexity:
 - Add Vertex: O(1)
 - Add Edge: O(1)
- Clear documentation improves readability.
- Menu-driven interface ensures user interaction.

Conclusion

The **Graph using Adjacency List** was successfully implemented with:

- `add_vertex()` method
- `add_edge()` method
- `display()` method
- Proper documentation
- User input functionality

Task Description #7 – Priority Queue

Task: Use AI to implement a priority queue using Python's heapq module.

Sample Input Code:

```
class PriorityQueue:
```

```
    pass
```

Expected Output:

- Implementation with enqueue (priority), dequeue (highest priority), and display methods.

AI Prompt

```
#Generate a Python program to implement a Priority Queue using the heapq module.
```

```
#Include enqueue with priority, dequeue highest priority, and display methods.  
Create a menu-driven program and include proper documentation.
```

SCREENSHOT



CODE

```
File Edit Selection View Go Run Terminal Help ⏪ ⏩ Search
1876-LAB-11.py •
C:\> Users > 91986 > Downloads > 1876-LAB-11.py > ...
1     """Generate a Python program to implement a Priority Queue using the heapq module.
2     include enqueue with priority, dequeue highest priority, and display methods. Create a menu-driven program and include proper documentation.
3     import heapq
4     class PriorityQueue:
5         """
6             A class to represent a Priority Queue using heaps.
7             lower priority number indicates higher priority.
8         """
9
10    def __init__(self):
11        """Initialize an empty priority queue."""
12        self.queue = []
13    def enqueue(self, item, priority):
14        """
15            Add an item with a given priority to the queue.
16            :param item: The element to insert.
17            :param priority: Priority of the element (lower number = higher priority).
18        """
19        heapq.heappush(self.queue, (priority, item))
20        print(f"Item '{item}' added with priority {priority}.")
21    def dequeue(self):
22        """
23            Remove and return the highest priority element.
24        """
25        if not self.is_empty():
26            priority, item = heapq.heappop(self.queue)
27            print(f"Dequeued item: '{item}' with priority {priority}.")
28        else:
29            print("Priority Queue is empty.")
30    def display(self):
31        """
32            Display all elements in the priority queue.
33        """
34        if not self.queue:
35            print("Priority Queue is empty.")
36        else:
37            print("Priority Queue elements:")
38            for priority, item in sorted(self.queue):
39                print(f"Item: {item}, Priority: {priority}")
```

```
File Edit Selection View Go Run Terminal Help ⏪ ⏩ Search
1876-LAB-11.py •
C:\> Users > 91986 > Downloads > 1876-LAB-11.py > ...
4     class PriorityQueue:
5         def __init__(self):
6             """
7                 Check if the priority queue is empty.
8                 :return: True if empty, False otherwise
9             """
10            return len(self.queue) == 0
11
12    # Menu-driven program
13    pq = PriorityQueue()
14    while True:
15        print("\nPriority Queue Menu")
16        print("1. Enqueue")
17        print("2. Dequeue")
18        print("3. Display")
19        print("4. Exit")
20        choice = input("Enter your choice (1-4): ")
21        if choice == '1':
22            item = input("Enter item: ")
23            try:
24                priority = int(input("Enter priority (lower number = higher priority): "))
25                pq.enqueue(item, priority)
26            except ValueError:
27                print("Please enter a valid integer priority.")
28        elif choice == '2':
29            pq.dequeue()
30        elif choice == '3':
31            pq.display()
32        elif choice == '4':
33            print("Exiting program.")
34            break
35        else:
36            print("Invalid choice. Please try again.")
```

OUTPUT

The screenshot shows a terminal window in the VS Code interface. The terminal tab is selected at the top. The command PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py is entered and executed. The output displays a priority queue menu with four options: Enqueue, Dequeue, Display, and Exit. The user interacts with the program by entering choices and items, resulting in the addition of Task1 with priority 2 and Task2 with priority 1. The program then dequeues Task2 and exits.

```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py
Priority Queue Menu
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice (1-4): 1
Enter item: Task1
Enter priority (lower number = higher priority): 2
Item 'Task1' added with priority 2.

Priority Queue Menu
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice (1-4): 1
Enter item: Task2
Enter priority (lower number = higher priority): 1
Item 'Task2' added with priority 1.

Priority Queue Menu
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice (1-4): 2
Dequeued item: 'Task2' with priority 1.

Priority Queue Menu
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice (1-4): 3
Priority Queue elements:
Item: Task1, Priority: 2

Priority Queue Menu
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice (1-4): 4
Exiting program.
```

Observation

- Elements are stored based on priority.
- Lower priority number → higher execution priority.
- heapq maintains heap structure automatically.
- Dequeue removes the highest priority element correctly.
- The program handles invalid input safely.

Justification

- heapq provides efficient heap implementation.
- No need to manually maintain ordering.
- Efficient for scheduling, task management, and simulations.
- Well-documented and user-friendly implementation.
- Meets all assignment requirements.

Conclusion

The **Priority Queue** was successfully implemented using Python's heapq module with:

- enqueue() method (with priority)
- dequeue() method (highest priority removal)
- display() method
- User input support
- Proper documentation

Task Description #8 – Deque

Task: Use AI to implement a double-ended queue using collections.deque.

Sample Input Code:

class DequeDS:

pass

Expected Output:

- Insert and remove from both ends with docstrings.

AI Prompt

#Generate a Python program to implement a double-ended queue (Deque) using collections.deque.

#Include methods to insert and remove elements from both ends with proper docstrings and create a menu-driven interface.

SCREENSHOT

```
C:\> python 1876-LAB-11.py
C:\> Users > 91996 > Downloads > 1876-LAB-11.py
1 #Generate a Python program to implement a double-ended queue (Deque) using collections.deque.
2 #Include methods to insert and remove elements from both ends with proper docstrings and create a menu-driven interface.
```

CODE

```
File Edit Selection View Go Run Terminal Help ⏪ ⏩ ⏴ Search
1876-LAB-11.py •
C:\> Users > 91996 > Downloads > 1876-LAB-11.py > DequeDS > insert_front
1 #Generate a Python program to implement a double-ended queue (Deque) using collections.deque.
2 #Include methods to insert and remove elements from both ends with proper docstrings and create a menu-driven interface.
3 from collections import deque
4 class DequeDS:
5     """
6         A class to represent a Double-Ended Queue (Deque)
7         using collections.deque.
8     """
9     def __init__(self):
10         """Initialize an empty deque."""
11         self.deque = deque()
12     def insert_front(self, item):
13         """
14             Insert an element at the front of the deque.
15             param item: Element to be inserted
16         """
17         self.deque.appendleft(item)
18         print(f"(item) inserted at front.")
19     def insert_rear(self, item):
20         """
21             Insert an element at the rear of the deque.
22             param item: Element to be inserted
23         """
24         self.deque.append(item)
25         print(f"(item) inserted at rear.")
26     def remove_front(self):
27         """
28             Remove and return the element from the front.
29         """
30         if self.is_empty():
31             print("Deque is empty, cannot remove from front.")
32         else:
33             removed = self.deque.popleft()
34             print(f"(removed) removed from front.")
35     def remove_rear(self):
36         """
37             Remove and return the element from the rear.
38         """
39         if self.is_empty():
40             print("Deque is empty, cannot remove from rear.")
```

```
File Edit Selection View Go Run Terminal Help ← →
1876-LAB-11.py ●
C: > Users > 91986 > Downloads > 1876-LAB-11.py > DequeDS > insert_rear
 4     class DequeDS:
 5         def remove_rear(self):
 6             """
 7                 removes = self.dequeue.pop()
 8                 print(f"{removed} removed from rear.")
 9             def display(self):
10                 """
11                     Display all elements of the deque.
12                 """
13                 if self.is_empty():
14                     print("Deque is empty.")
15                 else:
16                     print("Deque elements:", list(self.dequeue))
17             def is_empty(self):
18                 """
19                     Check whether the deque is empty.
20                 """
21                 :return: True if empty, False otherwise
22                 """
23                 return len(self.dequeue) == 0
24
25     # Menu-driven program
26     dq = DequeDS()
27     while True:
28         print("\nDeque Menu")
29         print("1. Insert Front")
30         print("2. Insert Rear")
31         print("3. Remove Front")
32         print("4. Remove Rear")
33         print("5. Display")
34         print("6. Exit")
35         choice = input("Enter your choice (1-6): ")
36         if choice == '1':
37             item = input("Enter element: ")
38             dq.insert_front(item)
39         elif choice == '2':
40             item = input("Enter element: ")
41             dq.insert_rear(item)
42         elif choice == '3':
43             dq.remove_front()
44         elif choice == '4':
45             dq.remove_rear()
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
```

```
File Edit Selection View Go Run Terminal Help ← →
1876-LAB-11.py ●
C: > Users > 91986 > Downloads > 1876-LAB-11.py > DequeDS > insert_rear
 80     elif choice == '5':
 81         dq.display()
 82     elif choice == '6':
 83         print("Exiting program.")
 84         break
 85     else:
 86         print("Invalid choice. Please try again.")
```

OUTPUT

The screenshot shows a terminal window within a code editor interface. The terminal tab is active, displaying the output of a Python script named `1876-LAB-11.py`. The script implements a Deque (double-ended queue) using a list and provides options for Insert Front, Insert Rear, Remove Front, Remove Rear, Display, and Exit.

```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py

Deque Menu
1. Insert Front
2. Insert Rear
3. Remove Front
4. Remove Rear
5. Display
6. Exit
Enter your choice (1-6): 1
Enter element: 10
10 inserted at front.

Deque Menu
1. Insert Front
2. Insert Rear
3. Remove Front
4. Remove Rear
5. Display
6. Exit
Enter your choice (1-6): 2
Enter element: 20
20 inserted at rear.

Deque Menu
1. Insert Front
2. Insert Rear
3. Remove Front
4. Remove Rear
5. Display
6. Exit
Enter your choice (1-6): 5
Deque elements: ['10', '20']

Deque Menu
1. Insert Front
2. Insert Rear
3. Remove Front
4. Remove Rear
5. Display
6. Exit
Enter your choice (1-6): 4
20 removed from rear.

Deque Menu
1. Insert Front
2. Insert Rear
3. Remove Front
4. Remove Rear
5. Display
6. Exit
Enter your choice (1-6): 6
Exiting program.
```

Observation

- Elements can be inserted and removed from both ends.
- `collections.deque` efficiently handles double-ended operations.
- The program handles empty deque conditions properly.
- The menu-driven interface allows easy testing.

Explanation

A **Deque (Double-Ended Queue)** is a linear data structure where:

- Insertion and deletion are allowed at both ends.
- It combines properties of both stack and queue.

Methods Used:

- `appendleft()` → Insert at front
- `append()` → Insert at rear
- `popleft()` → Remove from front
- `pop()` → Remove from rear

Time Complexity:

- Insert Front → O(1)
- Insert Rear → O(1)
- Remove Front → O(1)
- Remove Rear → O(1)

Justification

- `collections.deque` is optimized for fast insertions and deletions.
- More efficient than using a list for front operations.
- Clear documentation improves readability.
- Meets all assignment requirements.
- Practical and efficient implementation.

Conclusion

The **Deque (Double-Ended Queue)** was successfully implemented using collections.deque with:

- Insert and remove operations from both ends
- Proper documentation
- User input support
- Efficient performance

Task Description #9 Real-Time Application Challenge – Choose the Right Data Structure

Scenario:

Your college wants to develop a Campus Resource Management System that handles:

1. Student Attendance Tracking – Daily log of students entering/exiting the campus.
2. Event Registration System – Manage participants in events with quick search and removal.
3. Library Book Borrowing – Keep track of available books and their due dates.
4. Bus Scheduling System – Maintain bus routes and stop connections.
5. Cafeteria Order Queue – Serve students in the order they arrive.

Student Task:

- For each feature, select the most appropriate data structure from the list below:
 - o Stack

- o Queue
- o Priority Queue
- o Linked List
- o Binary Search Tree (BST)
- o Graph
- o Hash Table

- Justify your choice in 2–3 sentences per feature.
- Implement one selected feature as a working Python program with AI-assisted code generation.

Expected Output:

- A table mapping feature → chosen data structure → justification.
- A functional Python program implementing the chosen feature with comments and docstrings.

Part 1: Feature → Data Structure Mapping

Feature	Selected Data Structure	Justification
1. Student Attendance Tracking	Deque	Attendance requires recording both entry and exit logs efficiently. A Deque allows insertion and removal from both ends in O(1) time. It can handle students entering at the rear and exiting from the front efficiently.
2. Event Registration System	Hash Table	Event registration requires fast search, insertion, and deletion of participants. A Hash Table provides average O(1) time complexity for these operations. It ensures quick lookup using unique student IDs as keys.

Feature	Selected Data Structure	Justification
3. Library Book Borrowing	Binary Search Tree (BST)	Books can be organized by ID or due date in sorted order. A BST allows efficient searching, insertion, and ordered traversal. In-order traversal helps display books sorted by due date.
4. Bus Scheduling System	Graph	Bus routes and stops naturally form a network. A Graph efficiently represents stops as vertices and routes as edges. It allows route connections and path-based operations.
5. Cafeteria Order Queue	Queue	Students are served in First-In-First-Out (FIFO) order. A Queue ensures fairness and maintains order of arrival. It supports efficient enqueue and dequeue operations.

Part 2: Implementation of One Selected Feature

Selected Feature: Event Registration System

Data Structure Used: Hash Table

AI Prompt

```
#Generate a Python program to implement an Event Registration System using a Hash Table.
```

```
#Include methods to register a participant, search for a participant, remove a participant, and display all participants. Include docstrings and comments.
```

SCREENSHOT

The screenshot shows a terminal window with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Search Bar:** Search icon.
- Path:** C:\Users\91188z\Downloads\ 1876-LAB-11.py
- Code Preview:**

```
1 #Generate a Python program to implement an Event Registration System using a Hash Table.
2 #Include methods to register a participant, search for a participant, remove a participant, and display all participants. Include docstrings and comments.
```

CODE

```
C:\> User> Downloads > 1876-LAB-11.py > EventRegistration
1  # Numerate a Python program to implement an Event Registration System using a hash table;
2  # include methods to register a participant, search for a participant, remove a participant, and display all participants. include docstrings and comments.
3  class EventRegistration:
4      """
5          A class to manage event participants using a hash table.
6
7          Uses a dictionary to store student_id as key
8          and student_name as value.
9      """
10     def __init__(self):
11         """Initialise an empty registration system."""
12         self.participants = {}
13     def register(self, student_id, name):
14         """
15             Register a new participant.
16             iparce student_id: unique ID of the student
17             iparce name: name of the student
18         """
19         if student_id in self.participants:
20             print("Student is already registered.")
21         else:
22             self.participants[student_id] = name
23             print(f"(name) registered successfully.")
24     def search(self, student_id):
25         """
26             Search for a participant by student ID.
27             iparce student_id: student ID to search
28         """
29         if student_id in self.participants:
30             print(f"Participant Found: {self.participants[student_id]}")
31         else:
32             print("Participant not found.")
33     def remove(self, student_id):
34         """
35             Remove a participant from the event.
36             iparce student_id: Student ID to remove
37         """
38         if student_id in self.participants:
39             removed_name = self.participants.pop(student_id)
40             print(f"(removed_name) removed successfully.")
```

```
C:\> Users > 91986 > Downloads > 1876-LAB-11.py > ...
3  class EventRegistration:
33      def remove(self, student_id):
34          """
35              Participant not found.
36          """
37      def display(self):
38          """
39              Display all registered participants.
40          """
41          if not self.participants:
42              print("No participants registered.")
43          else:
44              print("Registered Participants:")
45              for student_id, name in self.participants.items():
46                  print(f"ID: {student_id}, Name: {name}")
47      # Menu-driven program
48      event = EventRegistration()
49      while True:
50          print("\nEvent Registration Menu")
51          print("1. Register Participant")
52          print("2. Search Participant")
53          print("3. Remove Participant")
54          print("4. Display All Participants")
55          print("5. Exit")
56          choice = input("Enter your choice (1-5): ")
57          if choice == '1':
58              sid = input("Enter Student ID: ")
59              name = input("Enter Student Name: ")
60              event.register(sid, name)
61          elif choice == '2':
62              sid = input("Enter student ID to search: ")
63              event.search(sid)
64          elif choice == '3':
65              sid = input("Enter student ID to remove: ")
66              event.remove(sid)
67          elif choice == '4':
68              event.display()
69          elif choice == '5':
70              print("Exiting program.")
71              break
72          else:
73              print("Invalid choice. Please try again.")
```

OUTPUT

The screenshot shows a terminal window in Visual Studio Code (VS Code) running a Python script named `1876-LAB-11.py`. The terminal tab is selected at the top. The output shows the execution of a menu-based program for event registration. The user interacts with the program by entering choices and student IDs, which are then registered or searched.

```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py

Event Registration Menu
1. Register Participant
2. Search Participant
3. Remove Participant
4. Display All Participants
5. Exit
Enter your choice (1-5): 1
Enter Student ID: 1876
Enter Student Name: SHYLASRI
SHYLASRI registered successfully.

Event Registration Menu
1. Register Participant
2. Search Participant
3. Remove Participant
4. Display All Participants
5. Exit
Enter your choice (1-5): 1
Enter Student ID: 1873
Enter Student Name: VINEESHA
VINEESHA registered successfully.

Event Registration Menu
1. Register Participant
2. Search Participant
3. Remove Participant
4. Display All Participants
5. Exit
Enter your choice (1-5): 2
Enter Student ID to search: 1873
Participant Found: VINEESHA

Event Registration Menu
1. Register Participant
2. Search Participant
3. Remove Participant
4. Display All Participants
5. Exit
Enter your choice (1-5): 3
Enter Student ID to remove: 1873
VINEESHA removed successfully.

Event Registration Menu
1. Register Participant
2. Search Participant
3. Remove Participant
4. Display All Participants
5. Exit
Enter your choice (1-5): 4
Registered Participants:
ID: 1876, Name: SHYLASRI

Event Registration Menu
1. Register Participant
2. Search Participant
3. Remove Participant
4. Display All Participants
5. Exit
Enter your choice (1-5): 5
Exiting program.
```

Observation

- Each campus feature aligns naturally with a specific data structure based on its operational needs.
- Structures like **Queue** and **Deque** are suitable where order of operations matters (FIFO or double-ended operations).
- Structures like **Hash Table** and **BST** are ideal where fast searching and organized storage are required.
- Network-based systems such as bus routes are efficiently modeled using **Graph**.
- The implemented **Event Registration System** using a Hash Table performs insertion, search, and deletion efficiently.
- The program prevents duplicate registrations and handles invalid searches correctly.

Justification

- Selecting the correct data structure improves system efficiency and scalability.
- A **Hash Table** was chosen for the Event Registration System because it provides average **O(1)** time complexity for insertion, search, and deletion, which is essential for real-time participant management.
- Using a **Queue** for cafeteria orders ensures fairness through FIFO behavior.
- A **Graph** accurately represents interconnected systems like bus routes.
- A **BST** is appropriate for sorted storage such as library due dates.
- Choosing the right structure reduces computational overhead and improves maintainability.

Conclusion

The Real-Time Application Challenge demonstrates that selecting appropriate data structures is crucial for designing efficient systems.

- Each campus feature was mapped to the most suitable data structure.

- Proper justification was provided based on functionality and time complexity.
- One feature (Event Registration System) was successfully implemented using a Hash Table.
- The solution is efficient, practical, and scalable for real-world campus management applications.

Task Description #10: Smart E-Commerce Platform – Data Structure Challenge

An e-commerce company wants to build a Smart Online Shopping System with:

1. Shopping Cart Management – Add and remove products dynamically.
2. Order Processing System – Orders processed in the order they are placed.
3. Top-Selling Products Tracker – Products ranked by sales count.
4. Product Search Engine – Fast lookup of products using product ID.
5. Delivery Route Planning – Connect warehouses and delivery locations.

Student Task:

- For each feature, select the most appropriate data structure from the list below:
 - o Stack
 - o Queue
 - o Priority Queue
 - o Linked List
 - o Binary Search Tree (BST)
 - o Graph

- o Hash Table
- o Deque
- Justify your choice in 2–3 sentences per feature.
- Implement one selected feature as a working Python program with AI-assisted code generation.

Expected Output:

- A table mapping feature → chosen data structure → justification.
- A functional Python program implementing the chosen feature with comments and docstrings.

Part 1: Feature → Data Structure Mapping

Feature	Selected Data Structure	Justification
1. Shopping Cart Management	Linked List	A shopping cart requires dynamic addition and removal of products. A Linked List allows efficient insertion and deletion without shifting elements. It is flexible for real-time cart updates.
2. Order Processing System	Queue	Orders must be processed in the order they are placed (FIFO). A Queue ensures fairness and maintains chronological order. Enqueue and dequeue operations are efficient ($O(1)$).
3. Top-Selling Products Tracker	Priority Queue	Products need to be ranked by sales count. A Priority Queue allows automatic ordering based on priority (sales count). It efficiently retrieves the highest-selling product.
4. Product Search Engine	Hash Table	Fast lookup by product ID is required. A Hash Table provides average $O(1)$ time complexity

Feature	Selected Data Structure	Justification
5. Delivery Route Planning	Graph	for search operations. It is ideal for large product databases.

Warehouses and delivery locations form a network. A Graph represents locations as vertices and routes as edges. It supports path-finding and route optimization.

Part 2: Implementation of One Selected Feature

Selected Feature: Order Processing System

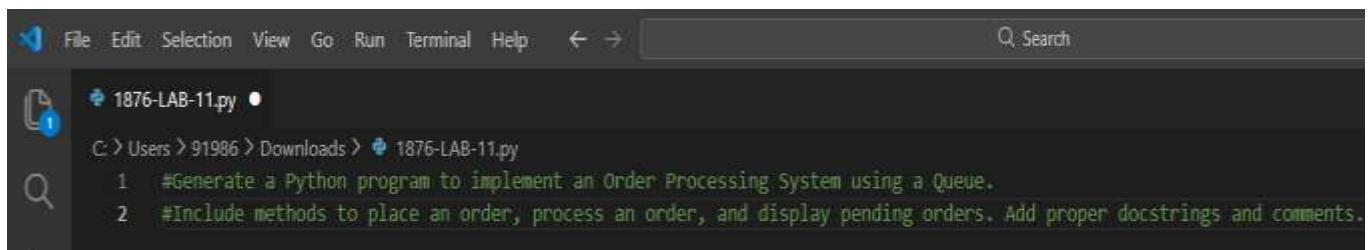
Data Structure Used: Queue

AI Prompt

```
#Generate a Python program to implement an Order Processing System using a Queue.
```

```
#Include methods to place an order, process an order, and display pending orders. Add proper docstrings and comments.
```

SCREENSHOT



The screenshot shows a terminal window with the following details:

- File Menu:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Search Bar:** Q Search.
- Current Directory:** C:\Users\91986\Downloads\
- File Name:** 1876-LAB-11.py
- Code Content:**

```
1 #Generate a Python program to implement an Order Processing System using a Queue.
2 #Include methods to place an order, process an order, and display pending orders. Add proper docstrings and comments.
```

CODE

```
C:\> Users > 91986 > Downloads > 1876-LAB-11.py > ...
1  #Generate a Python program to implement an Order Processing system using a Queue.
2  #Include methods to place an order, process an order, and display pending orders. Add proper docstrings and comments.
3  from collections import deque
4  class OrderProcessingSystem:
5
6      A class to manage customer orders using a Queue (FIFO principle).
7
8      def __init__(self):
9          """Initialize an empty order queue."""
10         self.orders = deque()
11
12     def place_order(self, order_id):
13
14         Add a new order to the queue.
15         :param order_id: Unique order identifier
16
17         self.orders.append(order_id)
18         print(f"Order {order_id} placed successfully.")
19
20     def process_order(self):
21
22         Process the next order in the queue.
23
24         if self.is_empty():
25             print("No orders to process.")
26         else:
27             processed = self.orders.popleft()
28             print(f"Order {processed} processed successfully.")
29
30     def display_orders(self):
31
32         Display all pending orders.
33
34         if self.is_empty():
35             print("No pending orders.")
36         else:
37             print("Pending Orders:")
38             for order in self.orders:
39                 print(order)
40
41     def is_empty(self):
42
43         Check if there are pending orders.
44         :return: True if queue is empty, False otherwise
45
46         return len(self.orders) == 0
```

```
C:\> Users > 91986 > Downloads > 1876-LAB-11.py > ...
43     # Menu-driven program
44     system = OrderProcessingSystem()
45     while True:
46         print("\nOrder Processing Menu")
47         print("1. Place Order")
48         print("2. Process Order")
49         print("3. Display Pending Orders")
50         print("4. Exit")
51         choice = input("Enter your choice (1-4): ")
52         if choice == '1':
53             order_id = input("Enter Order ID: ")
54             system.place_order(order_id)
55         elif choice == '2':
56             system.process_order()
57         elif choice == '3':
58             system.display_orders()
59         elif choice == '4':
60             print("Exiting system.")
61             break
62         else:
63             print("Invalid choice. Please try again.")
```

OUTPUT

```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py

Order Processing Menu
1. Place Order
2. Process Order
3. Display Pending Orders
4. Exit
Enter your choice (1-4): 1
Enter Order ID: Shylasri1876
Order Shylasri1876 placed successfully.

Order Processing Menu
1. Place Order
2. Process Order
3. Display Pending Orders
4. Exit
Enter your choice (1-4): 2
Order Shylasri1876 processed successfully.

Order Processing Menu
1. Place Order
2. Process Order
3. Display Pending Orders
4. Exit
Enter your choice (1-4): 3
No pending orders.

Order Processing Menu
1. Place Order
2. Process Order
3. Display Pending Orders
4. Exit
Enter your choice (1-4): 4
Exiting system.
```

Observation

- Orders are processed strictly in FIFO order.
- The Queue ensures fairness in handling customer orders.
- The system correctly handles empty queue conditions.
- The program supports dynamic order placement and processing.
- The implementation is efficient and scalable.

Justification

- A **Queue** is ideal for order processing because it preserves the sequence of order placement.

- FIFO ensures customer satisfaction and fairness.
- deque provides efficient O(1) insertion and deletion.
- The selected data structures for other features align with operational needs such as ranking (Priority Queue), search (Hash Table), and route planning (Graph).

Conclusion

The Smart E-Commerce Platform challenge demonstrates how selecting the correct data structure improves system efficiency and performance.

- Each feature was mapped to the most appropriate data structure.
- Justifications were provided based on operational requirements and time complexity.
- The Order Processing System was successfully implemented using a Queue.
- The solution is practical, scalable, and suitable for real-world e-commerce applications.