

Assignment-11.1

Data Structures with AI: Implementing Fundamental Structures

K. Santhosh Kumar

2403a51l21

B-51

Task Description #1 - Stack Implementation:

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

PROMPT: Generate a Python class for the Stack data structure using a list. Implement push, pop, peek, and is_empty methods following the LIFO principle. Add proper docstrings and handle empty stack errors appropriately.

Sample Input Code:

The screenshot shows a code editor window titled "AI ASSISTED CODING". In the center, there is a code template for a Stack class. The code includes methods for push, pop, peek, and is_empty, along with a main loop for user interaction. The code is as follows:

```
ASSIGN-11-1.py > ...
1  ## TASK-1: Generate a Python class for the Stack data structure using a list. Implement push, pop, peek, and is_empty methods
2  following the LIFO principle. Add proper docstrings and handle empty stack errors appropriately.
3
4  class Stack:
5      def __init__(self):
6          self.stack = []
7
8      def push(self, item):
9          self.stack.append(item)
10         print(f"{item} pushed into stack.")
11
12      def pop(self):
13          if not self.is_empty():
14              print(f"Popped element: {self.stack.pop()}")
15          else:
16              print("Stack is empty.")
17
18      def peek(self):
19          if not self.is_empty():
20              print(f"Top element: {self.stack[-1]}")
21          else:
22              print("Stack is empty.")
23
24      def is_empty(self):
25          return len(self.stack) == 0
26
27      def display(self):
28          print("Stack elements:", self.stack)
29
30  s = Stack()
31  while True:
32      print("1. Push")
33      print("2. Pop")
34      print("3. Peek")
35      print("4. Display")
36      print("5. Exit")
37      choice = input("Enter your choice: ")
38
39      if choice == '1':
40          value = input("Enter value to push: ")
41          s.push(value)
42      elif choice == '2':
43          s.pop()
44      elif choice == '3':
45          s.peek()
46      elif choice == '4':
47          s.display()
48      elif choice == '5':
49          print("Exiting program...")
50          break
51      else:
52          print("Invalid choice! Try again.")
```

The right side of the interface has a "CHAT" section with a message from "BLACKBOX Agent" and a "SESSIONS" section. The bottom status bar shows file information like "Ln 49, Col 44" and "Python 3.14.2".

The screenshot shows the same code editor window after the AI has generated the code. The code now includes additional logic for user input handling and error checking. The code is as follows:

```
ASSIGN-11-1.py > Stack
1
2  class Stack:
3      def __init__(self):
4          print("Stack elements:", self.stack)
5
6      def __str__(self):
7          return str(self.stack)
8
9      def push(self, value):
10         self.stack.append(value)
11         print(f"{value} pushed into stack.")
12
13      def pop(self):
14          if not self.is_empty():
15              value = self.stack.pop()
16              print(f"Popped element: {value}")
17          else:
18              print("Stack is empty.")
19
20      def peek(self):
21          if not self.is_empty():
22              print(f"Top element: {self.stack[-1]}")
23          else:
24              print("Stack is empty.")
25
26      def is_empty(self):
27          return len(self.stack) == 0
28
29      def display(self):
30          print("Stack elements:", self.stack)
31
32  s = Stack()
33  while True:
34      print("1. Push")
35      print("2. Pop")
36      print("3. Peek")
37      print("4. Display")
38      print("5. Exit")
39      choice = input("Enter your choice: ")
40
41      if choice == '1':
42          value = input("Enter value to push: ")
43          s.push(value)
44      elif choice == '2':
45          s.pop()
46      elif choice == '3':
47          s.peek()
48      elif choice == '4':
49          s.display()
50      elif choice == '5':
51          print("Exiting program...")
52          break
53      else:
54          print("Invalid choice! Try again.")
```

OUTPUT:

```
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 24
24 pushed into stack.

1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 2
```

EXPLANATION: A Stack is a linear data structure that follows the LIFO (Last In First Out) principle, where the last element inserted is the first one removed. Operations such as push, pop, and peek are performed at one end called the top. It is commonly used in function calls, undo operations, and expression evaluation.

Task Description #2 - Queue Implementation:

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

PROMPT: Create a Python class for a Queue using a list. Implement enqueue, dequeue, peek, and size methods following the FIFO principle. Add proper documentation and error handling.

Sample Input Code:

```
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 24
24 pushed into stack.

1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 2
```

```
q = Queue()
while True:
    print("\n1. Enqueue")
    print("2. Dequeue")
    print("3. Peek")
    print("4. Display")
    print("5. Exit")
    choice = input("Enter your choice: ")
    if choice == '1':
        value = input("Enter value to enqueue: ")
        q.enqueue(value)
    elif choice == '2':
        q.dequeue()
    elif choice == '3':
        q.peek()
    elif choice == '4':
        q.display()
    elif choice == '5':
        print("Exiting program...")
        break
    else:
        print("Invalid choice! Try again.")
```

OUTPUT:

```
PS C:\Users\sarik\OneDrive\Desktop\AI ASSISTED CODING & C:/Users/sarik/AppData/Local/Python/pythoncore-3.14-64/python.exe "c:/users/sarik/onedrive/Desktop/AI ASSISTED CODING/ASSIGN-11-1.py"
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to enqueue: 24
24 enqueued into queue.

1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 4
[Output: 24]
```

EXPLANATION: A Queue is a linear data structure that follows the FIFO (First In First Out) principle, where the first inserted element is removed first. Elements are added at the rear and removed from the front. It is widely used in scheduling systems, buffering, and real-world waiting line applications.

Task Description #3 - Linked List:

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

PROMPT: Generate a Python implementation of a Singly Linked List including a Node class and LinkedList class. Implement insert and display methods with clear documentation.

Sample Input Code:

```

101  #> TASK-3: Generate a Python implementation of a Singly Linked List including a Node class and LinkedList class. Implement
102  #> insert and display methods with clear documentation.
103  class Node:
104      def __init__(self, data):
105          self.data = data
106          self.next = None
107
108  class LinkedList:
109      def __init__(self):
110          self.head = None
111
112      def insert(self, data):
113          new_node = Node(data)
114          if self.head is None:
115              self.head = new_node
116              print(f"[{data}] inserted as head of the list.")
117          else:
118              current = self.head
119              while current.next:
120                  current = current.next
121              current.next = new_node
122              print(f"[{data}] inserted into the list.")
123
124      def display(self):
125          if self.head is None:
126              print("The linked list is empty.")
127          else:
128              current = self.head
129              elements = []
130              while current:
131                  elements.append(current.data)
132                  current = current.next
133              print(f"Linked List elements: {', '.join(map(str, elements))}")
134
135  ll = LinkedList()
136  while True:
137      print("\nll. Insert")
138      print("ll. Display")
139      print("ll. Exit")
140      choice = input("Enter your choice: ")
141      if choice == '1':
142          value = input("Enter value to insert: ")
143          ll.insert(value)
144      elif choice == '2':
145          ll.display()
146      elif choice == '3':
147          print("Exiting program...")
148      else:
149          print("Invalid choice! Try again.")

```

OUTPUT:

```

PS C:\Users\sarik\OneDrive\Desktop\AI ASSISTED CODING & C:/Users/sarik/AppData/Local/Python/pythoncore-3.14.64/python.exe "c:/users/sarik/onedrive/Desktop/AI ASSISTED CODING/ASSIGN-11-1.py"
1. Insert
2. Display
3. Exit
Enter your choice: 1
Enter value to insert: 11
11 inserted as head of the list.

1. Insert
2. Display
3. Exit
Enter your choice: 1
Enter value to insert: 14
14 inserted into the list.

1. Insert
2. Display
3. Exit
Enter your choice: 1
Enter value to insert: 24
24 inserted into the list.

```

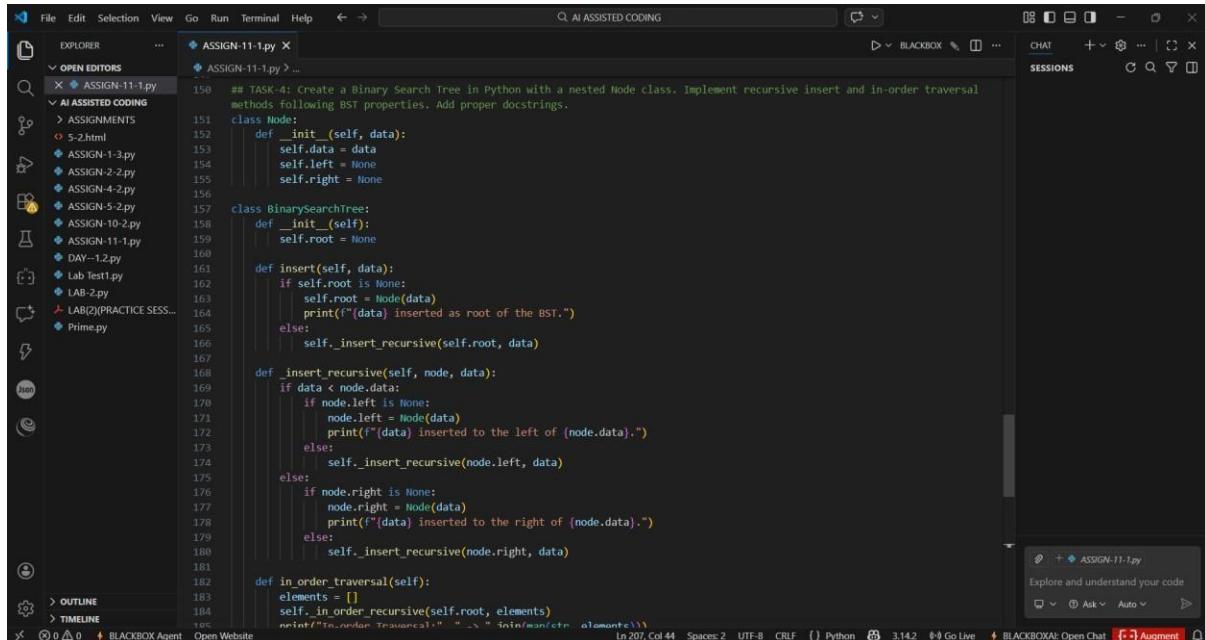
EXPLANATION: A Singly Linked List consists of nodes where each node contains data and a reference to the next node. Unlike arrays, it does not require contiguous memory, making it dynamic in size. It allows efficient insertions and deletions compared to fixed-size structures.

Task Description #4 - Binary Search Tree (BST):

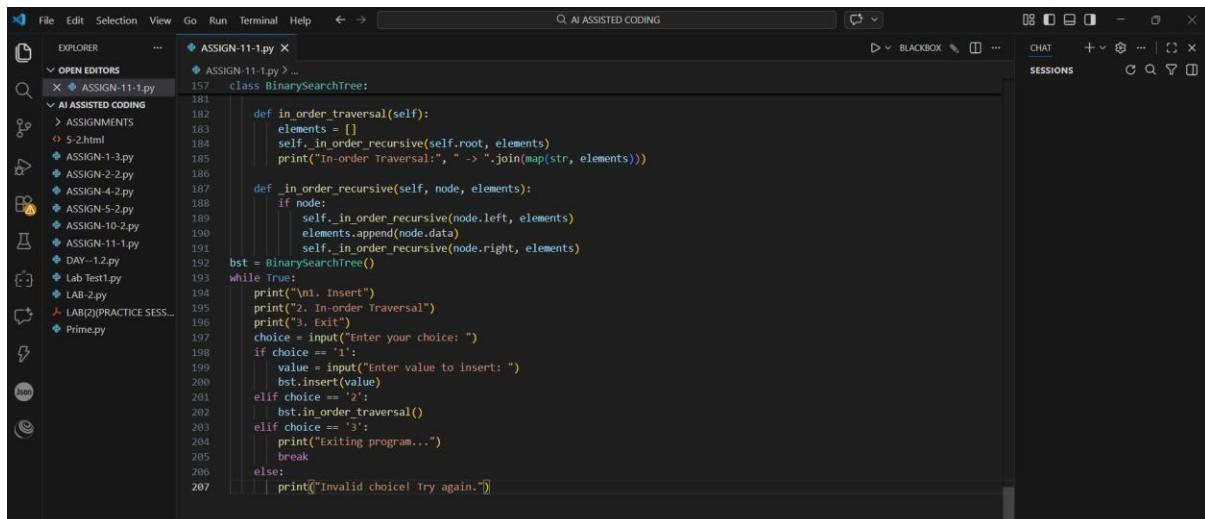
Task: Use AI to create a BST with insert and in-order traversal methods.

PROMPT: Create a Binary Search Tree in Python with a nested Node class. Implement recursive insert and in-order traversal methods following BST properties. Add proper docstrings.

Sample Input Code:

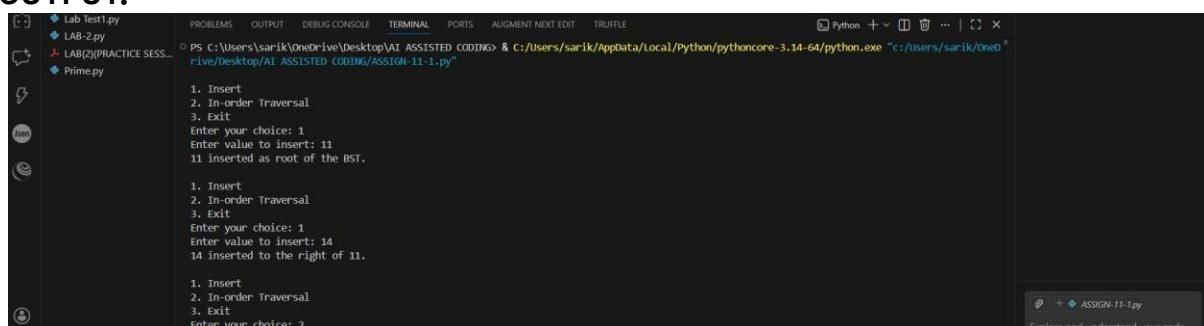


```
150 ## TASK-4: Create a Binary Search Tree in Python with a nested Node class. Implement recursive insert and in-order traversal
151 class Node:
152     def __init__(self, data):
153         self.data = data
154         self.left = None
155         self.right = None
156
157     class BinarySearchTree:
158         def __init__(self):
159             self.root = None
160
161         def insert(self, data):
162             if self.root is None:
163                 self.root = Node(data)
164                 print(f"{data} inserted as root of the BST.")
165             else:
166                 self._insert_recursive(self.root, data)
167
168         def _insert_recursive(self, node, data):
169             if data < node.data:
170                 if node.left is None:
171                     node.left = Node(data)
172                     print(f"{data} inserted to the left of {node.data}.")
173                 else:
174                     self._insert_recursive(node.left, data)
175
176             if node.right is None:
177                 node.right = Node(data)
178                 print(f"{data} inserted to the right of {node.data}.")
179             else:
180                 self._insert_recursive(node.right, data)
181
182         def in_order_traversal(self):
183             elements = []
184             self._in_order_recursive(self.root, elements)
185             print("In-order Traversal: " + " ".join(map(str, elements)))
186
187             def _in_order_recursive(self, node, elements):
188                 if node:
189                     self._in_order_recursive(node.left, elements)
190                     elements.append(node.data)
191                     self._in_order_recursive(node.right, elements)
192
193         bst = BinarySearchTree()
194         while True:
195             print("1. Insert")
196             print("2. In-order Traversal")
197             print("3. Exit")
198             choice = input("Enter your choice: ")
199             if choice == '1':
200                 value = input("Enter value to insert: ")
201                 bst.insert(value)
202             elif choice == '2':
203                 bst.in_order_traversal()
204             elif choice == '3':
205                 print("Exiting program...")
206                 break
207             else:
208                 print("Invalid choice! Try again.")
```



```
157     class BinarySearchTree:
158         def __init__(self):
159             self.root = None
160
161         def in_order_traversal(self):
162             elements = []
163             self._in_order_recursive(self.root, elements)
164             print("In-order Traversal: " + " ".join(map(str, elements)))
165
166         def _in_order_recursive(self, node, elements):
167             if node:
168                 self._in_order_recursive(node.left, elements)
169                 elements.append(node.data)
170                 self._in_order_recursive(node.right, elements)
171
172         bst = BinarySearchTree()
173         while True:
174             print("1. Insert")
175             print("2. In-order Traversal")
176             print("3. Exit")
177             choice = input("Enter your choice: ")
178             if choice == '1':
179                 value = input("Enter value to insert: ")
180                 bst.insert(value)
181             elif choice == '2':
182                 bst.in_order_traversal()
183             elif choice == '3':
184                 print("Exiting program...")
185                 break
186             else:
187                 print("Invalid choice! Try again.")
```

OUTPUT:



```
PS C:\Users\sarik\OneDrive\Desktop\AI ASSISTED CODING> & C:/Users/sarik/AppData/Local/Python/pythoncore-3.14-64/python.exe "c:/Users/sarik/OneD
rive/Desktop/AI ASSISTED CODING/ASSIGN-11-1.py"
1. Insert
2. In-order Traversal
3. Exit
Enter your choice: 1
Enter value to insert: 11
11 inserted as root of the BST.

1. Insert
2. In-order Traversal
3. Exit
Enter your choice: 1
Enter value to insert: 14
14 inserted to the right of 11.

1. Insert
2. In-order Traversal
3. Exit
Enter your choice: 2
```

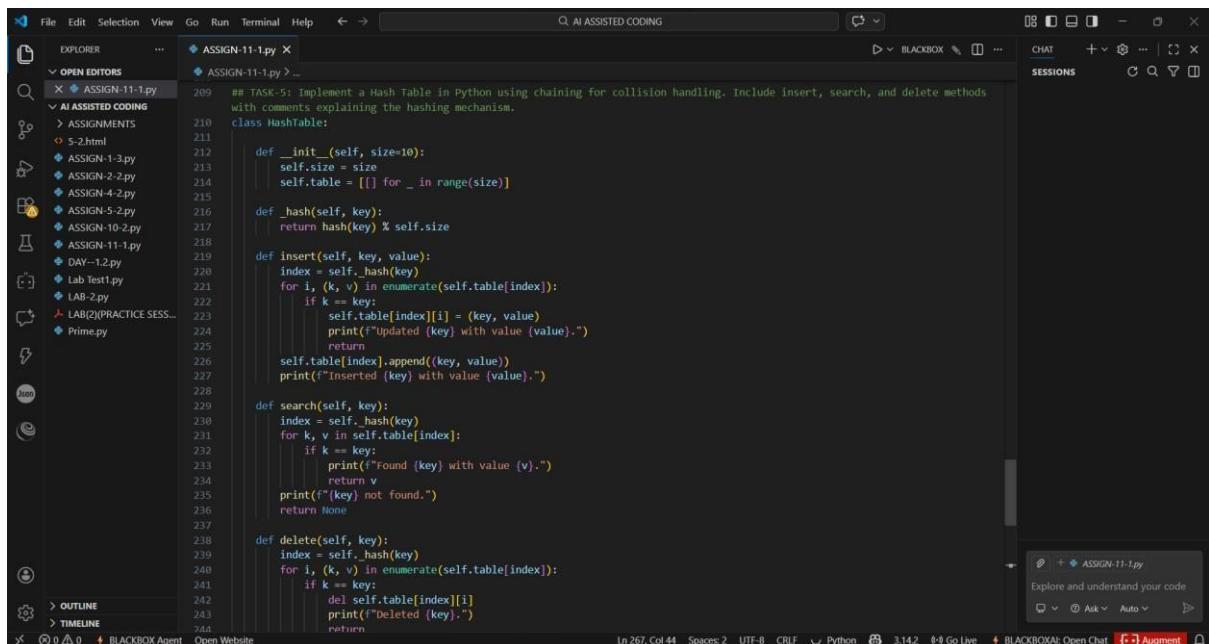
EXPLANATION: A Binary Search Tree is a hierarchical data structure where the left child contains smaller values and the right child contains larger values than the root. This property makes searching, insertion, and deletion efficient. In-order traversal of a BST produces sorted output.

Task Description #5 - Hash Table:

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

PROMPT: Implement a Hash Table in Python using chaining for collision handling. Include insert, search, and delete methods with comments explaining the hashing mechanism.

Sample Input Code:



```

File Edit Selection View Go Run Terminal Help < > O AI ASSISTED CODING
EXPLORER OPEN EDITORS AI ASSISTED CODING
ASSIGN-11-1.py ...
ASSIGN-11-1.py ...
209 ## TASK-5: Implement a Hash Table in Python using chaining for collision handling. Include insert, search, and delete methods
with comments explaining the hashing mechanism.
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267

def __init__(self, size=10):
    self.size = size
    self.table = [[ ] for _ in range(size)]

def _hash(self, key):
    return hash(key) % self.size

def insert(self, key, value):
    index = self._hash(key)
    for i, (k, v) in enumerate(self.table[index]):
        if k == key:
            self.table[index][i] = (key, value)
            print(f"Updated {key} with value {value}.")
            return
    self.table[index].append((key, value))
    print(f"Inserted {key} with value {value}.")

def search(self, key):
    index = self._hash(key)
    for k, v in self.table[index]:
        if k == key:
            print(f"Found {key} with value {v}.")
            return v
    print(f"{key} not found.")
    return None

def delete(self, key):
    index = self._hash(key)
    for i, (k, v) in enumerate(self.table[index]):
        if k == key:
            del self.table[index][i]
            print(f"Deleted {key}.")
            return
    print(f"{key} not found for deletion.")

ht = HashTable()
while True:
    print("1. Insert")
    print("2. Search")
    print("3. Delete")
    print("4. Exit")
    choice = input("Enter your choice: ")
    if choice == '1':
        key = input("Enter key to insert: ")
        value = input("Enter value to insert: ")
        ht.insert(key, value)
    elif choice == '2':
        key = input("Enter key to search: ")
        ht.search(key)
    elif choice == '3':
        key = input("Enter key to delete: ")
        ht.delete(key)
    elif choice == '4':
        print("Exiting program...")
        break
    else:
        print("Invalid choice! Try again.")
```

File Edit Selection View Go Run Terminal Help < > O AI ASSISTED CODING
EXPLORER OPEN EDITORS AI ASSISTED CODING
ASSIGN-11-1.py ...
ASSIGN-11-1.py ...
209 ## TASK-5: Implement a Hash Table in Python using chaining for collision handling. Include insert, search, and delete methods
with comments explaining the hashing mechanism.
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267

def __init__(self, size=10):
 self.size = size
 self.table = [[] for _ in range(size)]

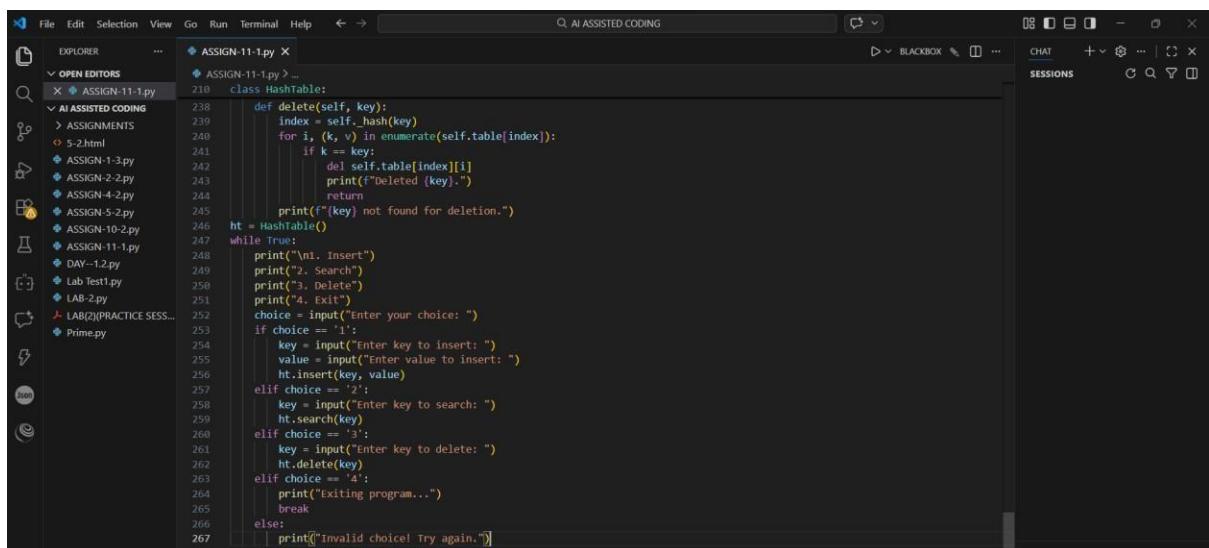
def _hash(self, key):
 return hash(key) % self.size

def insert(self, key, value):
 index = self._hash(key)
 for i, (k, v) in enumerate(self.table[index]):
 if k == key:
 self.table[index][i] = (key, value)
 print(f"Updated {key} with value {value}.")
 return
 self.table[index].append((key, value))
 print(f"Inserted {key} with value {value}.")

def search(self, key):
 index = self._hash(key)
 for k, v in self.table[index]:
 if k == key:
 print(f"Found {key} with value {v}.")
 return v
 print(f"{key} not found.")
 return None

def delete(self, key):
 index = self._hash(key)
 for i, (k, v) in enumerate(self.table[index]):
 if k == key:
 del self.table[index][i]
 print(f"Deleted {key}.")
 return
 print(f"{key} not found for deletion.")

ht = HashTable()
while True:
 print("1. Insert")
 print("2. Search")
 print("3. Delete")
 print("4. Exit")
 choice = input("Enter your choice: ")
 if choice == '1':
 key = input("Enter key to insert: ")
 value = input("Enter value to insert: ")
 ht.insert(key, value)
 elif choice == '2':
 key = input("Enter key to search: ")
 ht.search(key)
 elif choice == '3':
 key = input("Enter key to delete: ")
 ht.delete(key)
 elif choice == '4':
 print("Exiting program...")
 break
 else:
 print("Invalid choice! Try again.")



```

File Edit Selection View Go Run Terminal Help < > O AI ASSISTED CODING
EXPLORER OPEN EDITORS AI ASSISTED CODING
ASSIGN-11-1.py ...
ASSIGN-11-1.py ...
209 ## TASK-5: Implement a Hash Table in Python using chaining for collision handling. Include insert, search, and delete methods
with comments explaining the hashing mechanism.
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267

def __init__(self, size=10):
    self.size = size
    self.table = [[ ] for _ in range(size)]

def _hash(self, key):
    return hash(key) % self.size

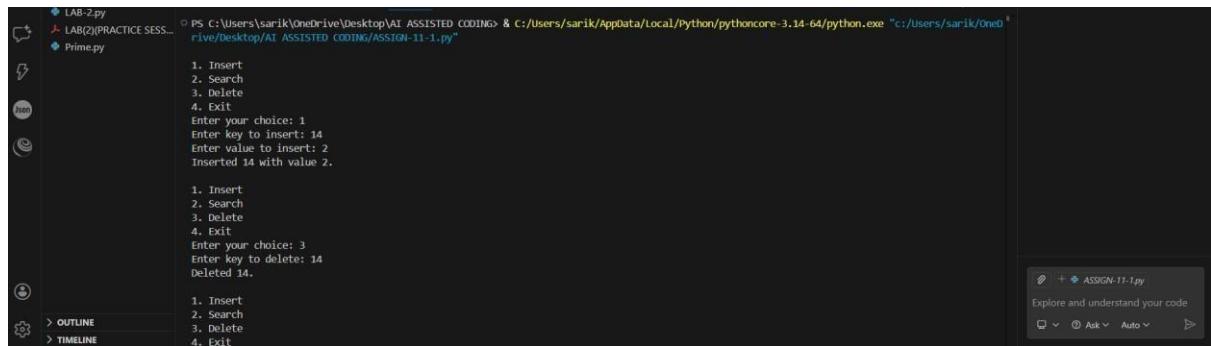
def insert(self, key, value):
    index = self._hash(key)
    for i, (k, v) in enumerate(self.table[index]):
        if k == key:
            self.table[index][i] = (key, value)
            print(f"Updated {key} with value {value}.")
            return
    self.table[index].append((key, value))
    print(f"Inserted {key} with value {value}.")

def search(self, key):
    index = self._hash(key)
    for k, v in self.table[index]:
        if k == key:
            print(f"Found {key} with value {v}.")
            return v
    print(f"{key} not found.")
    return None

def delete(self, key):
    index = self._hash(key)
    for i, (k, v) in enumerate(self.table[index]):
        if k == key:
            del self.table[index][i]
            print(f"Deleted {key}.")
            return
    print(f"{key} not found for deletion.")

ht = HashTable()
while True:
    print("1. Insert")
    print("2. Search")
    print("3. Delete")
    print("4. Exit")
    choice = input("Enter your choice: ")
    if choice == '1':
        key = input("Enter key to insert: ")
        value = input("Enter value to insert: ")
        ht.insert(key, value)
    elif choice == '2':
        key = input("Enter key to search: ")
        ht.search(key)
    elif choice == '3':
        key = input("Enter key to delete: ")
        ht.delete(key)
    elif choice == '4':
        print("Exiting program...")
        break
    else:
        print("Invalid choice! Try again.")
```

OUTPUT:



```
LAB-2.py
LAB(2)PRACTICE SESSION
Prime.py

1. Insert
2. Search
3. Delete
4. Exit
Enter your choice: 1
Enter key to insert: 14
Enter value to insert: 2
Inserted 14 with value 2.

1. Insert
2. Search
3. Delete
4. Exit
Enter your choice: 3
Enter key to delete: 14
Deleted 14.

1. Insert
2. Search
3. Delete
4. Exit
```

EXPLANATION: A Hash Table stores data in key-value pairs using a hash function to compute an index. It provides fast average-case time complexity for search, insertion, and deletion operations. Collisions are handled using techniques such as chaining.

Task Description #6 - Graph Representation:

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

PROMPT: Generate a Graph implementation in Python using an adjacency list representation. Include methods to add vertices, add edges, and display connections with documentation.

Sample Input Code:

OUTPUT:

The screenshot shows a code editor interface with two main panes. The top pane displays the Python code for a graph implementation, while the bottom pane shows the terminal output of running the code.

Code Editor Content (Top Pane):

```
## TASK-6: Generate a Graph implementation in Python using an adjacency list representation. Include methods to add vertices, add edges, and display connections with documentation.

class Graph:
    def __init__(self):
        self.adjacency_list = {}

    def add_vertex(self, vertex):
        if vertex not in self.adjacency_list:
            self.adjacency_list[vertex] = []
            print(f"Vertex {vertex} added.")
        else:
            print(f"Vertex {vertex} already exists.")

    def add_edge(self, vertex1, vertex2):
        if vertex1 in self.adjacency_list and vertex2 in self.adjacency_list:
            self.adjacency_list[vertex1].append(vertex2)
            self.adjacency_list[vertex2].append(vertex1)
            print(f"Edge added between {vertex1} and {vertex2}.")
        else:
            print("One or both vertices do not exist.")

    def display_connections(self):
        for vertex, edges in self.adjacency_list.items():
            print(f"\n{vertex}: {edges}")

g = Graph()
while True:
    print("1. Add Vertex")
    print("2. Add Edge")
    print("3. Display Connections")
    print("4. Exit")
    choice = input("Enter your choice: ")
    if choice == '1':
        vertex = input("Enter vertex to add: ")
        g.add_vertex(vertex)
    elif choice == '2':
        vertex1 = input("Enter first vertex: ")
        vertex2 = input("Enter second vertex: ")
        g.add_edge(vertex1, vertex2)
    elif choice == '3':
        g.display_connections()
    elif choice == '4':
        print("Exiting program...")
        break
    else:
        print("Invalid choice! Try again.")
```

Terminal Output (Bottom Pane):

```
PS C:\Users\sarik\OneDrive\Desktop\AI ASSISTED CODING> & c:/users/sarik/appdata/local/python/pythoncore-3.14-64/python.exe "c:/users/sarik/onedrive/Desktop/AI ASSISTED CODING/ASSIGN-11-1.py"
1. Add Vertex
2. Add Edge
3. Display Connections
4. Exit
Enter your choice: 1
Enter vertex to add: 14
Vertex 14 added.

1. Add Vertex
2. Add Edge
3. Display Connections
4. Exit
Enter your choice: 2
Enter first vertex: 11
Enter second vertex: 14
One or both vertices do not exist.
```

EXPLANATION: A Graph is a non-linear data structure used to represent relationships between entities. It consists of vertices (nodes) and edges (connections). Graphs are commonly used in networks, maps, and routing systems.

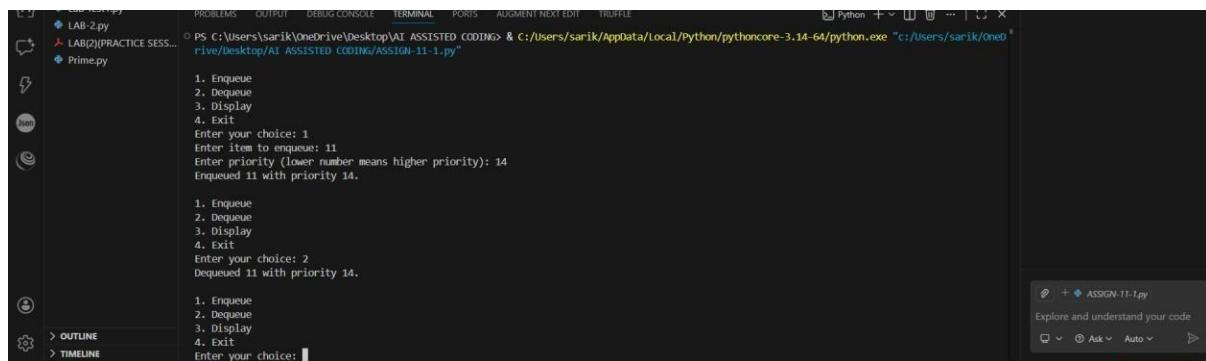
Task Description #7 - Priority Queue:

OUTPUT:

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

PROMPT: Create a Priority Queue in Python using the heapq module. Implement enqueue with priority, dequeue (highest priority first), and display methods. Add proper documentation.

Sample Input Code:

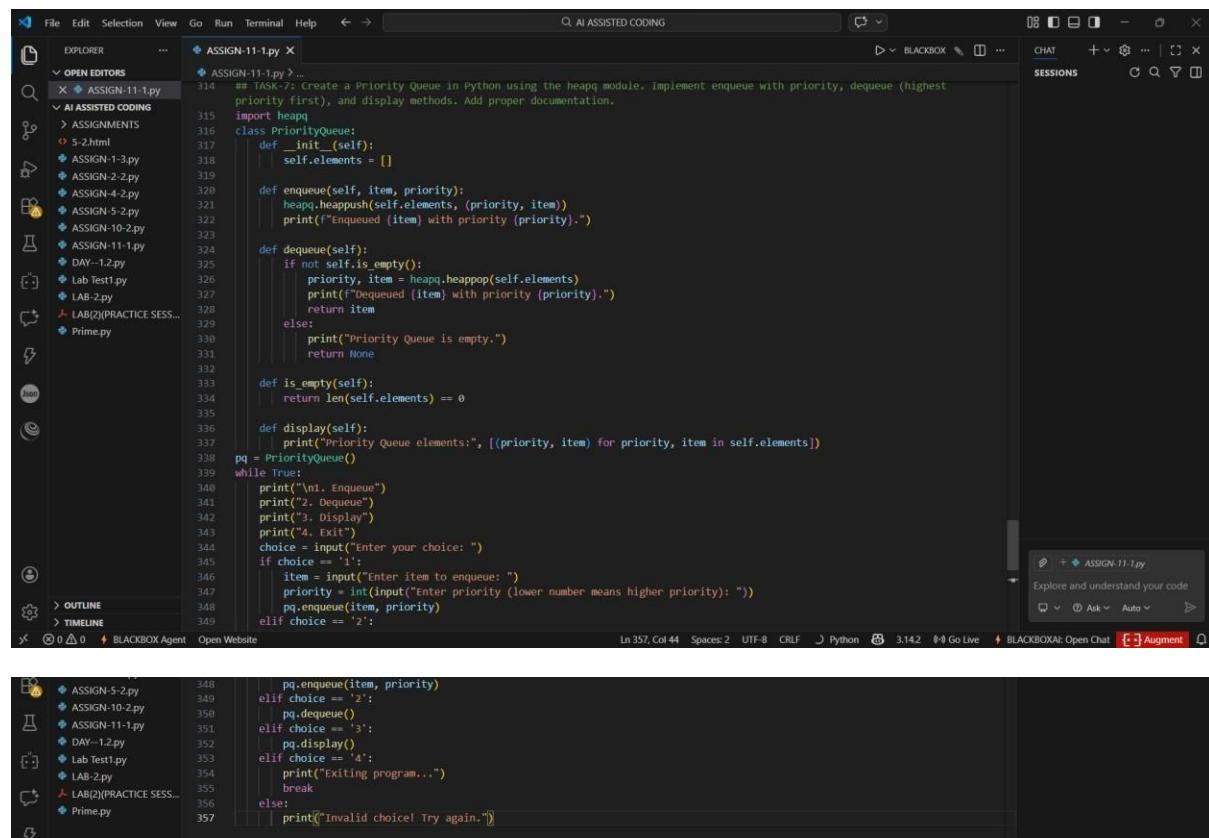


```
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter item to enqueue: 11
Enter priority (lower number means higher priority): 14
Enqueued 11 with priority 14.

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
Dequeued 11 with priority 14.

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
```

EXPLANATION: A Priority Queue is a special type of queue where elements are removed based on priority rather than order of insertion. Higher priority elements are processed first. It is typically implemented using a heap for efficiency.



```
## TASK-7: Create a Priority Queue in Python using the heapq module. Implement enqueue with priority, dequeue (highest priority first), and display methods. Add proper documentation.

class PriorityQueue:
    def __init__(self):
        self.elements = []

    def enqueue(self, item, priority):
        heapq.heappush(self.elements, (priority, item))
        print(f"Enqueued {item} with priority {priority}.")

    def dequeue(self):
        if not self.is_empty():
            priority, item = heapq.heappop(self.elements)
            print(f"Dequeued {item} with priority {priority}.")
            return item
        else:
            print("Priority Queue is empty.")
            return None

    def is_empty(self):
        return len(self.elements) == 0

    def display(self):
        print("Priority Queue elements:", [(priority, item) for priority, item in self.elements])

pq = PriorityQueue()
while True:
    print("\n1. Enqueue")
    print("2. Dequeue")
    print("3. Display")
    print("4. Exit")
    choice = input("Enter your choice: ")
    if choice == '1':
        item = input("Enter item to enqueue: ")
        priority = int(input("Enter priority (lower number means higher priority): "))
        pq.enqueue(item, priority)
    elif choice == '2':
        pq.dequeue()
    elif choice == '3':
        pq.display()
    elif choice == '4':
        print("Exiting program...")
        break
    else:
        print("Invalid choice! Try again.")
```

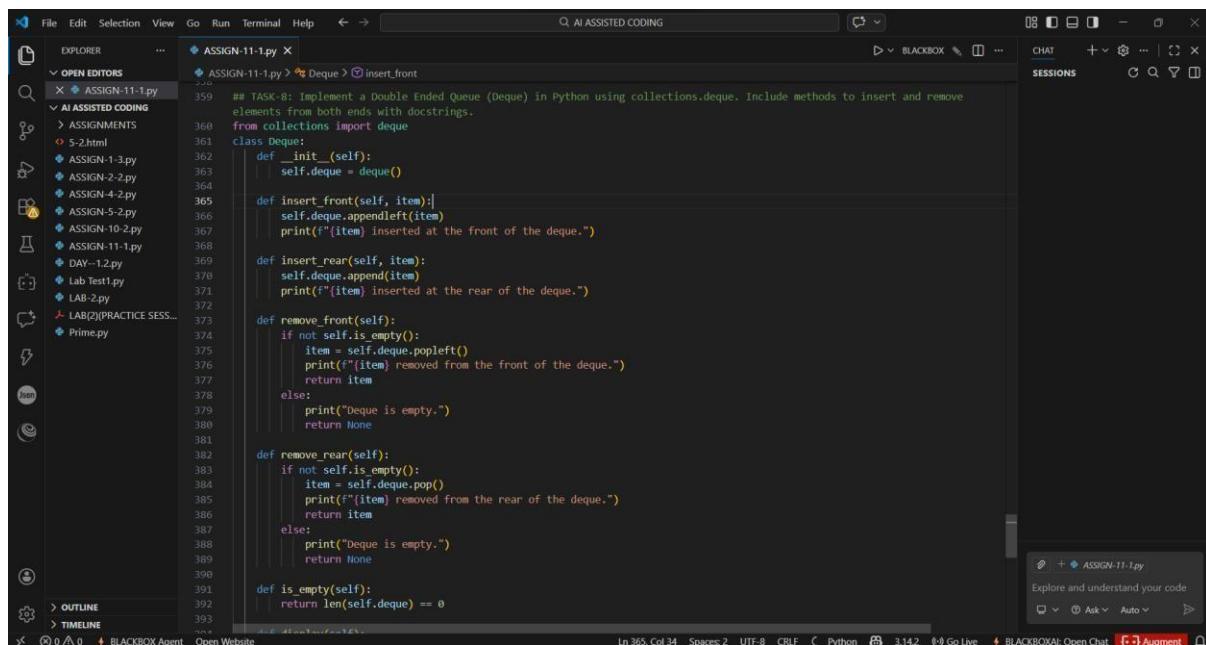
OUTPUT:

Task Description #8 - Deque:

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

PROMPT: Implement a Double Ended Queue (Deque) in Python using collections.deque. Include methods to insert and remove elements from both ends with docstrings.

Sample Input Code:



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

- File Path:** EXPLORER / OPEN EDITORS / AI ASSISTED CODING / ASSIGN-11-1.py
- Code Content:** Python code for a Deque class using collections.deque. The code includes methods for inserting and removing elements from both ends, each with a docstring. The code is numbered from 359 to 393.

```
359 ## TASK-8: Implement a Double Ended Queue (Deque) in Python using collections.deque. Include methods to insert and remove elements from both ends with docstrings.
360 from collections import deque
361
362 class Deque:
363     def __init__(self):
364         self.deque = deque()
365
366     def insert_front(self, item):
367         self.deque.appendleft(item)
368         print(f'{item} inserted at the front of the deque.')
369
370     def insert_rear(self, item):
371         self.deque.append(item)
372         print(f'{item} inserted at the rear of the deque.')
373
374     def remove_front(self):
375         if not self.is_empty():
376             item = self.deque.popleft()
377             print(f'{item} removed from the front of the deque.')
378             return item
379         else:
380             print("Deque is empty.")
381             return None
382
383     def remove_rear(self):
384         if not self.is_empty():
385             item = self.deque.pop()
386             print(f'{item} removed from the rear of the deque.')
387             return item
388         else:
389             print("Deque is empty.")
390             return None
391
392     def is_empty(self):
393         return len(self.deque) == 0
```

- Editor Tools:** Includes tabs for CHAT, SESSIONS, and various file operations.
- Bottom Status Bar:** Shows file path (ASSIGN-11-1.py), line count (Ln 365, Col 34), and other system information.

```
ASSIGN-11-1.py
393     def display(self):
394         print("Deque elements:", list(self.deque))
395
396     d = Deque()
397     while True:
398         print("\n1. Insert Front")
399         print("2. Insert Rear")
400         print("3. Remove Front")
401         print("4. Remove Rear")
402         print("5. Display")
403         print("6. Exit")
404
405         choice = input("Enter your choice: ")
406
407         if choice == '1':
408             item = input("Enter item to insert at front: ")
409             d.insert_front(item)
410         elif choice == '2':
411             item = input("Enter item to insert at rear: ")
412             d.insert_rear(item)
413         elif choice == '3':
414             d.remove_front()
415         elif choice == '4':
416             d.remove_rear()
417         elif choice == '5':
418             d.display()
419         elif choice == '6':
420             print("Exiting program...")
421         else:
422             print("Invalid choice! Try again.")
```

OUTPUT:

```
C:\> PS C:\Users\sarik\OneDrive\Desktop\AI ASSISTED CODING> & c:/users/sarik/appdata/local/python/pythoncore-3.14-64/python.exe "c:/users/sarik/oneo/LAB(2)/PRACTICE SESSIONS/ASSIGN-11-1.py"
1. Insert Front
2. Insert Rear
3. Remove Front
4. Remove Rear
5. Display
6. Exit
Enter your choice: 1
Enter item to insert at front: 11
11 inserted at the front of the deque.

1. Insert Front
2. Insert Rear
3. Remove Front
4. Remove Rear
5. Display
6. Exit
Enter your choice: 2
Enter item to insert at rear: 14
14 inserted at the rear of the deque.
```

EXPLANATION: A Deque (Double Ended Queue) allows insertion and deletion of elements from both the front and rear ends. It combines features of both stacks and queues. It is useful in applications like sliding window algorithms and task scheduling.